# Finding the Needle in the Heap: Combining Binary Analysis Techniques to Trigger Use-After-Free

Josselin Feist
Supervisors: Marie-Laure Potet and Laurent Mounier

March 29, 2017

# Plan

1. Context: Vulnerabilities
   - Program Vulnerabilities
   - Automated Program Analysis
2. Use-After-Free, a Complex Vulnerability
3. Using Static Analysis to Detect Use-After-Free
4. Using Dynamic Symbolic Execution to Trigger Use-After-Free
5. Thesis Contributions and Conclusion

# Context

## Security

- Recent interest from media and general public for computer security;
- Topic: security of computer programs
  - Programs contain bugs;
  - Bugs range from benign (almost no impact) to critical;
  - Some bugs decrease the security of the system → called **vulnerabilities**.
- Dirty Cow [Goo], Stagefright [Fin], Heartbleed [McM], ...

# Vulnerabilities

## Different categories of vulnerabilities

- Bad use of cryptography;
- Unsanitized inputs, such as SQL Injection;
- **Memory corruptions** $\sim$ low-level vulnerabilities.

## Possible consequences of memory corruption

- The system becomes unavailable (e.g., *denial-of-service* attacks);
- Critical information is leaked (such as cryptographic keys);
- The full system is compromised (unauthorized code execution).

# Who Has Interest in Vulnerabilities?

## Who uses them?

- Malicious individuals / groups (malware market, ...)
- Governments, industrial theft, ...

## Who finds them?

- Developers or internal researchers of vendors;
- *Bug-bounties* programs;
- Black market → prices can reach millions of dollars.

## Practical topic

Vulnerabilities research requires engineering process:

- Need for **working solutions**;
- **PoC** and **reproducibility** are important.

# Finding Vulnerabilities

Manual or **assisted** by automated program analysis techniques

## Dynamic analysis

- Fuzzing: generating inputs stressing the program (AFL, Radamsa, ..);
- Simple but efficient $\rightarrow$ **widely used** in industry;
- Not well-adapted to detect complex patterns.

## Static analysis

- Analyzing several behaviors of a program without executing it;
- Historically used for verification (proof the absence of bugs);
- **Less used** in security community (number of **false alarms**, no PoC).

## Other techniques

Distinction between dynamic and static not always pertinent:

- Symbolic execution (SAGE [GLM12], Mayhem [CARB12], ..);
- Guided fuzzing (Libfuzzer);
- Combining techniques [ZC10, BMMS11, HSNB13, BCDK14].

## Need for binary analysis

- Source code not available;
- Undefined behaviors;
- Precise memory layout.

→ Binaries analysis is harder than source code analysis, but it is mandatory in several contexts.

# Vulnerability Research: Past, Present, Future

## Past

- Buffer overflow, string format;
- Lots of manual efforts to find and exploit vulnerabilities;
- Individual hackers, few professionals.

## Present

- Overflow hard to exploit $\rightarrow$ use-after-free, type confusion;
- Better tools: *Smart* fuzzers, etc.;
- Business model of entire companies.

## Future

- Darpa CGC $\rightarrow$ fully-automated system;
- New attack vectors (Rowhammer [SD15]);
- Looking for protection killing class of vuln. or exploits.

# Plan

# Use-After-Free: Example

```c
1 char *login, *password;
2 login = malloc(..);
3 ...
4 free(login);
5 // login still points to the address returned by malloc
```

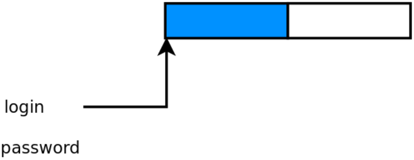



Figure: At line 2

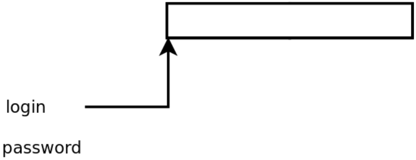
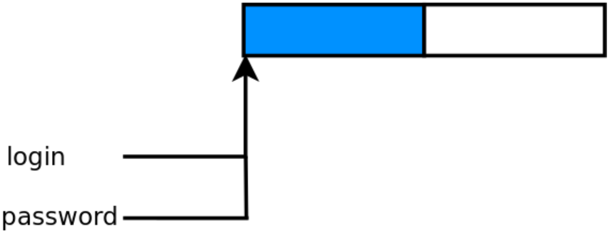



Figure: At line 5

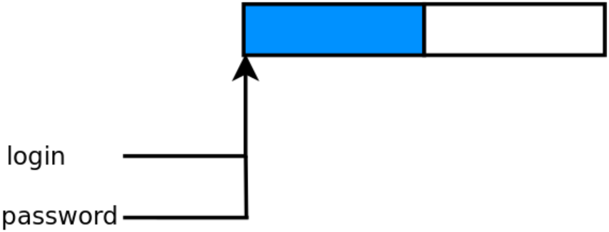
# Use-After-Free: Example

```c
char *login, *password;
login = malloc(..);
...
free(login);
// login still points to the address returned by malloc
password = malloc(..);
```



# Use-After-Free: Example

```c
char *login, *password;
login = malloc(..);
...
free(login);
// login still points to the address returned by malloc
password = malloc(..);
printf("Login: %s\n", login); // prints the password
```



# Definitions

## Dangling pointer

Pointer referencing a free block or a block reallocated to another pointer.

```
char *login;
login = malloc(..);
free(login);
```

## Use-After-Free

Use of a dangling pointer.

```
char *login;
login = malloc(..);
free(login);
[....];
printf("..", login);
```

## Dangerousness

- Reallocation of memory area used by $p$ ?
- Can lead to code execution, ...

# A Recent Vulnerability

- Recent vulnerability → less studied;
- Demonstrated to be dangerous: Operation Aurora [Wik13], Pwn2Own, ...

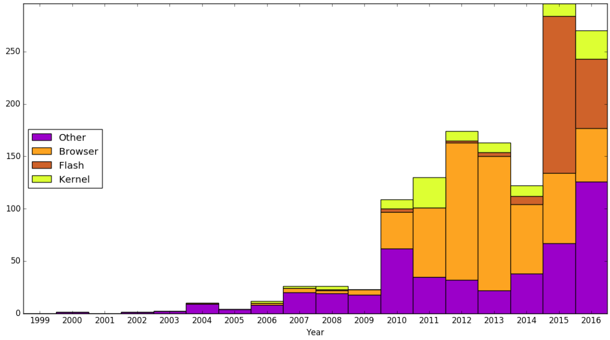



Figure: CVE: number of Use-After-Free (2016-12-28).

# Particularities of Use-After-Free

- Three events:
  - Allocation,
  - Free,
  - Use;
- Events can be distant in the code → need for **scalability**;
- Pointers are complex to track (e.g., *aliases* problem, ...) → need for **precision**;
- No easy "pattern" (like for buffer overflow / string format).

# Detecting Use-After-Free – State of the Art Methods

## Problems with existing methods

- Not sufficient alone
  - Dynamic detectors, such as ASan [SBPV12], Valgrind [Val], need other techniques to trigger the path;
- Not applicable in a security perspective
  - Static analyzers with too many false alarms and no PoC;
  - Need for manual annotations;
- Academic papers without tools available → no fair comparison;
- Industrial tools without description.

# Thesis Goals and Challenges

- **Apply formal methods** to **security purpose**;
- Develop techniques precise enough to **detect** and **trigger Use-After-Free**;
- Apply static analysis to **real-world binaries**:
  - Scalable analysis;
  - Programs designed without security in mind;
  - Avoiding no realistic working hypotheses.

→ Finding which methods can be applied in real contexts;
→ Finding the right trade-off between precision and scalability.

# Thesis Results

- Developing a static analysis unsound, but well-suited to detect **Use-After-Free on binaries**;
- Using dynamic symbolic execution to **remove false alarms** of the static analysis and to **generate PoC**;
- Found several **previously unknown** Use-After-Free in software;
- Open-source tool-chain.

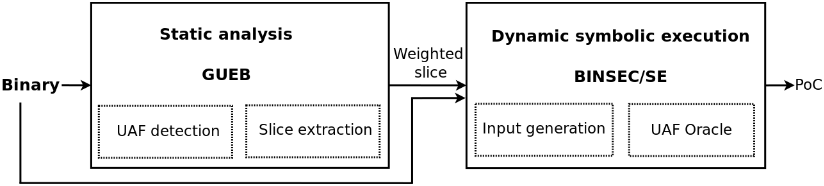
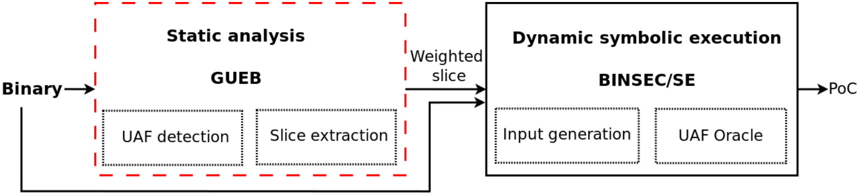



Figure: Architecture of our approach.

# Plan


Static analysis
GUEB
Binary
UAF detection
Slice extraction
Weighted slice
Dynamic symbolic execution
BINSEC/SE
Input generation
UAF Oracle
PoC

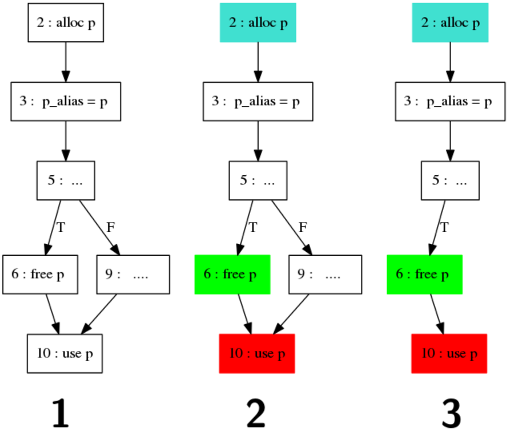
# Statically Detecting Use-After-Free

## Steps

1. Value analysis $\rightarrow$ tracking use of pointers,
2. Characterization of Use-After-Free,
3. Slice extraction.

# The Three Steps

```
...
 p=malloc(..);
 p_alias=p;
 ..
 if(..){
  free(p);
 }
 else{ ..}
 ..
 *p_alias = 42;
```



# Value Set Analysis (VSA)

Binary code = no variables, only memory and registers accesses.

## VSA: background [BR10]

- For each point of program, represent all possible memory states;
- *Transfer functions* → the transitions between instructions;
- Loop / recursion → compute fixed points (*costly*).

```
1  a = 1;
2  if(cond){
3   a = 2;
4  }
5  a = a + 1;
```

```
1   mov eax,1
2   jne L1
3   mov eax,2
4  L1:
5   add eax,1
```

| Inst | Memory state |
|---|---|
| 1: mov eax,1 | eax $\in$ [1] |
| 3: mov eax,2 | eax $\in$ [2] |
| 5: add eax,1 | eax $\in$ [2,3] |

# Our Value Analysis

## Detection requirements

- Track the use of pointers / aliases;
- Find the state of the heap objects (allocated / freed);
- Find paths leading to Use-After-Free.

## Light VSA

- *Best-effort* to track values;
- Unroll loops and inline functions;
- One allocation = one **new** memory area.

# Our Memory Model

## Overview

- For each instruction → **AbsEnv**;
- *AbsEnv* associates to each memory location **memLoc** a set of possible values **valueSet**;

- Example of *memLoc*:
  - $eax$,
  - $[esp_0 - 4]$: local variable,
  - $[chunk_0 + 0]$: heap chunk.
- Example of *valueSet*:
  - $chunk_0 + \{0, 4, 8\}$.

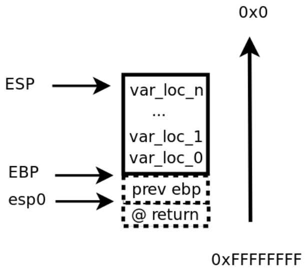



Figure: Local variables.

# Memory Model: Example

```
1  p=0; // p is accessed through the inital value of esp0 -4
2  if(cond){
3   p=malloc(); // malloc returns chunk0 + {0}
4  }
5  *p = .. // p = malloc or p = 0
```

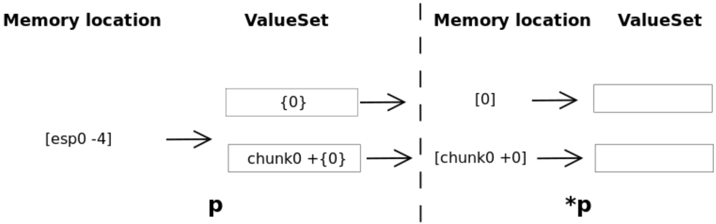



Figure: *AbsEnv* at line 5.

# Memory Model: Allocation Status

How to represent the status (allocated / freed) of a chunk?

## Three solutions proposed

- Object-Based: status kept apart of the memory model (using two functions: $HA$ and $HF$);
- Pointer-Based: status kept into the *valueSet*;
- A last variant allows to detect *Stack-based* Use-After-Free.

## Details

- Conservative free;
- Object-Based is more classic;
- Pointer-Based has better precision when path conditions are ignored.

# Allocation Status: Limitation of Object-Based

Limitations when dealing with path conditions

```
1 int *p=malloc(sizeof(int));
2 if(cond){
3  free(p);
4  p=malloc(sizeof(int));
5 }
6 // union of memory states
```

| Code | $Init_{reg}$ | Heap State |
|---|---|---|
| `1: p=malloc()` | $(esp_0 - 0x4) \rightarrow chunk_0$ | $HA = \{chunk_0\}$<br>$HF = \emptyset$ |
| `3: free(p)` | $(esp_0 - 0x4) \rightarrow chunk_0$ | $HA = \emptyset$<br>$HF = \{chunk_0\}$ |
| `4: p=malloc()` | $(esp_0 - 0x4) \rightarrow chunk_1$ | $HA = \{chunk_1\}$<br>$HF = \{chunk_0\}$ |
| `6: // union` | $(esp_0 - 0x4) \rightarrow chunk_0, chunk_1$ | $HA = \{chunk_1\}$<br>$HF = \{chunk_0\}$ |

Table: VSA results using the *object-based* representation.

# Pointer-Based Representation

```
int *p=malloc(sizeof(int));
if(cond){
 free(p);
 p=malloc(sizeof(int));
}
// union of memory states
```

| Code | $\text{Init}_{\text{reg}}$ |
|---|---|
| `1: p=malloc()` | $(esp_0 - 0x4) \rightarrow (chunk_0, A)$ |
| `3: free(p)` | $(esp_0 - 0x4) \rightarrow (chunk_0, F)$ |
| `4: p=malloc()` | $(esp_0 - 0x4) \rightarrow (chunk_1, A)$ |
| `6: // union` | $(esp_0 - 0x4) \rightarrow (chunk_0, A), (chunk_1, A)$ |

Table: VSA results using the *pointer-based* representation.

→ **No dangling pointer is kept.**

# Allocation Status: Comparison

## Discussion

- Benchmarks show that:
  - Pointer-based **reduce by two** the number of false alarms;
  - Pointer-based comes with no time or space overhead;
- If the analysis does not handle path conditions, pointer-based is to be used.

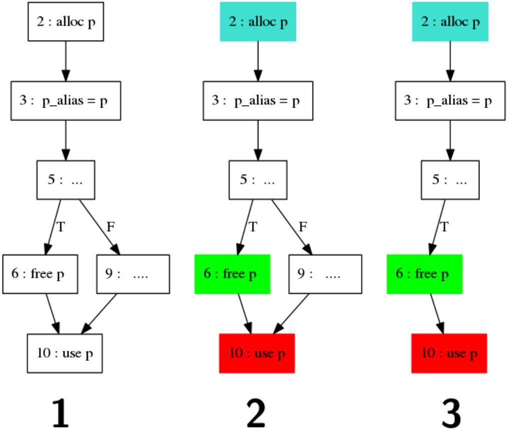
# Use-After-Free Characterization and Detection

## VSA results

- For each instruction $\rightarrow$ the memory state;
- For each heap element $\rightarrow$ allocated or freed;
- Use-After-Free characterization – checking the status of the heap elements on memory dereferencing:
  - load src, dst $\rightarrow$ $mem(src)$ is freed ?
  - store src, dst $\rightarrow$ $mem(dst)$ is freed ?
- A slice representing all paths going through the three events is extracted.

# Detecting Use-After-Free

```
...
 p=malloc(..);
 p_alias=p;
 ..
 if(..){
  free(p);
 }
 else{ ..}
 ..
 *p_alias = 42;
```



# Application to Real-World Binary Code

## VSA handling real binary code

- Handling nested and irreducible loops;
- Dealing with errors in the CFG;
- Heuristics adapting the VSA:
  - Function without return statement,
  - Lost of the stack frame,
  - ...
- Discussion on the validity of our VSA:
  - Results **inconsistent** with reality, yet **well-suited in practice**.

# Static Binary Analysis: Validity

| Origin | Type |
|---|---|
| No handling of conditions | Over-approximation |
| Weak updates | Over-approximation |
| Unrolling (missing paths) | Under-approximation |
| Inlining bounded by size | Under-approximation |
| Inlining bounded by depth | Under-approximation |
| Aliases between uninitialized values | Under-approximation |
| Inlining bounded by depth | Inconsistency |
| Recursion | Inconsistency |
| Incorrect CFG | Inconsistency |
| No overlap between memory location | Inconsistency |
| Unrolling (invalid paths) | Inconsistency |
| Ignored updates | Inconsistency |

Table: Summary of types of approximations for static analysis.

# Suitable Results

## Help analyzing the results

- A same dangling pointer can be used at multiples locations;
  - Solution: **grouping similar** Use-After-Free;
- Due to inlining, a function $f$ containing a Use-After-Free is reported as many times it is called;
  - Solution: **signature to detect likely-similar** Use-After-Free.

# Experiments

## Implementation

- GUEB: https://github.com/montyly/gueb
- Ocaml
- Use with IDA/BinNavi (REIL as intermediate representation).

## Questions

- Is GUEB precise enough to detect unknown Use-After-Free without raising too many false alarms?
- Is GUEB robust enough to be applied at scale?
- Two experimental results:
  - Finding previously unknown vulnerabilities in 6 software;
  - GUEB applied to a significant number of binaries (488).

# Previously Unknown Use-After-Free Found

| Name | #Lines | Time | #UAF | #Signature | #REIL ins. |
|---|---|---|---|---|---|
| alsabat | $\sim$ 2000 | $7s$ | 1 | 1 | 99933 |
| gnome-nettool | $\sim$ 6500 | $17s$ | 7 | 5 | 260882 |
| gifcolor (CVE-2016-3177) | $\sim$ 9000 | $21s$ | 15 | 12 | 233303 |
| jasper (CVE-2015-5221) | $\sim$ 34200 | $4m23$ | 255 | 114 | 2154927 |
| accel-ppd | $\sim$ 61000 | $5m5$ | 35 | 30 | 3907862 |
| openjpeg (CVE-2015-887) | $\sim$ 205200 | $6m10$ | 329 | 300 | 2170081 |

# Experiments

## Discussion

- GUEB yields an *acceptable* number of results;
- Largest (300) takes time to analyze, but it still realistic;
- In practice iterative analyze, several false alarms are removed by adding user-provided stubs to GUEB.

# Experiments: Robustness

## Results

- 488 binaries from */usr/bin* of Ubuntu 16.04,
- 406 without Use-After-Free found,
- 82 with Use-After-Free found. Most likely to be false alarms,
- Recall: too many false alarms is worse than missing Use-After-Free.

| # Bin | Signature ($\bar{x}$,med,max) | UAF ($\bar{x}$,med,max) | Time Total ($\bar{x}$,med,max) |
|---|---|---|---|
| 82 | $(9, 3, 210)$ | $(16, 5, 247)$ | 0h 52m 48s (38s,27s,4m 27s) |

# Static Analysis: Contributions

## Contributions

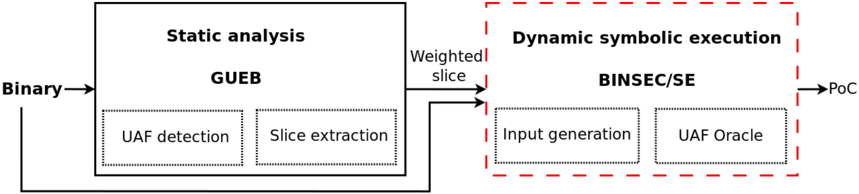
- Memory model and VSA well adapted to binary code and scalability;
- Study of heap objects modeling and UaF detection;
- Implementation and benchmarks demonstrating its efficiency and robustness.

## Limitations

- Too large programs,
- Multi-threading,
- Not well suited for some programs (e.g., *reference counters*).

$\rightarrow$ An analyst is still needed. Could we automatize more? Generating PoC?

# Plan


Static analysis
GUEB
UAF detection
Slice extraction
Binary
Weighted slice
Dynamic symbolic execution
BINSEC/SE
Input generation
UAF Oracle
PoC

# Dynamic Symbolic Execution (DSE)

DSE: an automated input generation technique.

## How it works

- From a program and an input, generate an execution trace;
- Build a *path predicate* from the trace = logical formula representing the trace as a set of constraints over the inputs;
- Use SMT solver to invert conditional instruction;
- If SAT, generate new inputs.

# DSE: Example

- First input: $a = 0$, generating a first trace;
- Build the path predicate;
- Invert condition at line 3 $\rightarrow a_0 \wedge a_1 = a_0 + 1 \wedge a_1 == 0x42$;
- Solve the formula: $a_0 = 0x41$, new input reaching line 4.

```
1  void f(int a){
2   a = a+1;
3   if(a == 0x42){
4    printf("Win!\n");
5   }
6  }
```

| Trace | Path Predicate |
|---|---|
| 1: int a | a0 ^ |
| 2: a = a +1 | a1 = a0 + 1 ^ |
| 3: a == 0x42 | a1 != 0x42 |
| 5: .. | |

# DSE

## Large recent interest in security

- Academic & Industrial interest;
- SAGE, KLEE, Mayhem, Angr, Triton, etc.;
- Young topic, still a lot of limitations;
- Other use: deobfuscation [DB16], etc.

## Challenges for path exploration

- Path explosion problem:
  - $\rightarrow$ Importance of the **exploration strategy**;
- Tuning on path predicate:
  - Concretization,
  - Inputs functions,
  - Libraries.

# Guided DSE

Guided DSE: Using a score function to prioritize the exploration.

## Our approach: Weigthed-Slice

- Guided DSE toward the Use-After-Free slice;
- 3 events (alloc / free / use) $\rightarrow$ 3 scores;
- DSE guided by the *next event to reach*;
- Partition nodes in a trace according if an event occurred.

We called our solution: *Weighted-Slice*.

# Weighted-Slice Guided DSE: Example

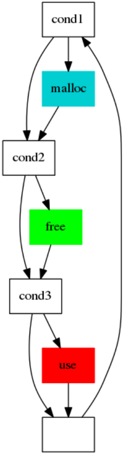



Figure: Slice

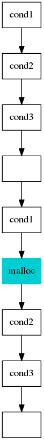



Figure: Trace $t$

# Weighted-Slice Guided DSE

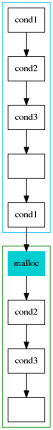



Figure: Trace with partition

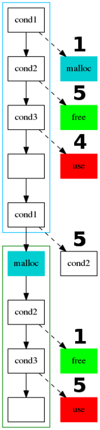



Figure: Trace with scores

# Detecting Use-After-Free Without False Alarms

## Detecting Use-After-Free on a trace

- Not so trivial;
- Trouble with indirect aliases.

```
p=malloc(..); // return X
free(p);      // free(X)
p2=malloc();  // return X
*p2 = 0;      // [X] = 0
*p = 0;       // [X] = 0
```

## Solution

- Need data dependencies, allocator modifications or metadata;
- Proposition: Oracle based on symbolic execution.

# Validity of the DSE

- DSE correctness: yes $\rightarrow$ paths generated are feasible;
- DSE completeness: no $\rightarrow$ bounded exploration;
- Oracle correctness: yes $\rightarrow$ Use-After-Free detected are true positives;
- Oracle completeness: no $\rightarrow$ Use-After-Free can be not detected.

- We produce true positive without false positives;
- Use-After-Free can be missed.

# Experiment

## Implementation

- Guided DSE implemented into the BINSEC/SE platform;
- Design of a generic exploration mechanism;
- Shortest path to the destination.

## Questions

- Is DSE exploration working on real-world examples?
- Is the engine able to trigger Use-After-Free?
- Is GUEB helping the exploration to trigger the Use-After-Free?
- Experimental result:
  - The method triggers the Use-After-Free of the `JasPer` CVE.

# Experiment

- Tested on `JasPer`, on a slice we knew to contain a Use-After-Free, with as seed a file filed with $'A'$;
- Comparison with standard exploration algorithm and fuzzers.

| Name | Time | UAF found | # Paths |
|---|---|---|---|
| WS-Guided (gueb) | $20m$ | **Yes** | 9 |
| *DFS (slice)* | $6h$ | *No* | 68 |
| *DFS* | $6h$ | *No* | 354 |
| *AFL* | $7h$ | *No* | 174 |
| *Radamsa* | $7h$ | *No* | $N/A$ |
| *AFL* (better seed file) | $< 1min$ | **Yes** | $< 10$ |
| *Radamsa* (better seed file) | $< 1min$ | **Yes** | $< 10$ |

# Experiment: Discussion and Limitations

- Only our solution found the Use-After-Free without a seed;
- Fuzzers found the vulnerability with a proper seed (only few mutations are needed in this case);
- Promising preliminary results, but:
  - A larger experiment is needed to validate the approach;
  - BINSEC/SE is still young, only `JasPer` was running properly;
  - The slice containing the Use-After-Free was explored $\rightarrow$ all the slices ($\sim$ 200) have to be explored, but it is still realistic.

# Dynamic Symbolic Execution: Conclusion

## Contributions

- How to guide DSE toward Use-After-Free using GUEB;
- Creation of a *proof-of-concept* on `JasPer`;
- Implementation into the BINSEC platform;
- Tuning on DSE:
  - Initial memory state;
  - Exploration enhancement based on programming patterns.

# Plan

1. Context: Vulnerabilities
2. Use-After-Free, a Complex Vulnerability
3. Using Static Analysis to Detect Use-After-Free
4. Using Dynamic Symbolic Execution to Trigger Use-After-Free
5. Thesis Contributions and Conclusion
    - Approach Discussion
    - Contributions and Perspectives

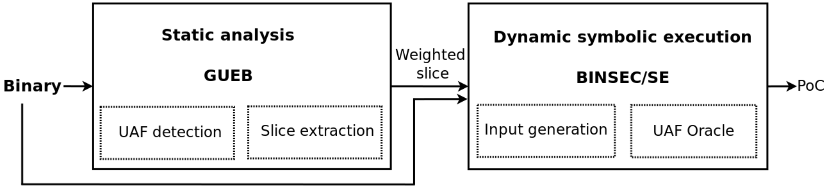



Figure: Architecture of our approach.

# Combining Static Analysis with DSE: Validity of the results

| GUEB | Results | BINSEC/SE | Results |
|---|---|---|---|
| Paths analyzed | Feasible, infeasible and unexplored paths | Concrete traces | All paths are feasible |
| | | Bounded exploration | Unexplored paths |
| UaF detected | True positives and | Oracle | True positives and no false positives |
| | False positives | Bounded exploration | False negatives |
| UaF not detected | False negatives | Not explored | False negatives |

Table: Validity of the global approach.

# Contributions

## Static analysis

- Design of a memory model and a VSA well suited to be applied on real-world binary code;
- Study of the heap model;
- Use-After-Free characterization and representation to provide suitable results.

## Dynamic symbolic execution

- Exploration algorithm using information provided by static analysis;
- Refinements on the DSE exploration and path predicates computation.

# Contributions

## Implementation and experiments

- Discovery of 6 new vulnerabilities;
- Creation of a PoC;
- All tools are open-source;
- All files are available to reproduce the experiments.

→ First **end-to-end** approach targeting Use-After-Free.

# Perspectives

## Static analysis

- Dedicated static analysis, less scalable, but more precise $\rightarrow$ reduce false alarms;
- Target other vulnerabilities (Use-Before-Initialization [LWP+17]), or specific software (e.g., kernel).

## Dynamic symbolic execution

- Testing the guided DSE on other programs;
- Combining with fuzzers [SGS+16];
- Slices from other origins (e.g., BinDiff on patched programs).

## Use-After-Free

- Study of the exploitability.

# Thesis Publications

- Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting Use-after-Free on binary code.
  2013 JCVHT;
- Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Using static analysis to detect use-after-free on binary code.
  In *1st Symposium on Digital Trust in Auvergne*, 2014;
- Josselin Feist. Gueb : Static detection of use-after-free on binary.
  In *ToorCon San Diego*, 2015;
- Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Guided dynamic symbolic execution using subgraph control-flow information.
  In *SEFM*, Lecture Notes in Computer Science, pages 76–81. Springer, 2016;
- Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and

  Marie-Laure Potet. Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free.
  In *6th Software Security, Protection, and Reverse Engineering Workshop, SSPREW 2016, Los Angeles, CA, USA, December 5-6, 2016*, pages 1–12, 2016.

# References I


Sebastien Bardin, Omar Chebaro, Mickal Delahaye, and Nikolai Kosmatov.
An all-in-one toolkit for automated white-box testing.
In Martina Seidl and Nikolai Tillmann, editors, *TAP*, volume 8570 of *Lecture Notes in Computer Science*, pages 53–60. Springer, 2014.

Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song.
Statically-directed dynamic automated test generation.
In Matthew B. Dwyer and Frank Tip, editors, *ISSTA*, pages 12–22. ACM, 2011.

Gogul Balakrishnan and Thomas Reps.
Wysinwyx: What you see is not what you execute.
*ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley.
Unleashing mayhem on binary code.
In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

Robin David and Sebastien Bardin.
Code deobfuscation: Intertwining dynamic, static and symbolic approaches.
*Black Hat Europe*, 2016.

Josselin Feist.
Gueb : Static detection of use-after-free on binary.
In *ToorCon San Diego*, 2015.

# References II


Jon Fingas.
Stagefright exploit reliably attacks android phones.
https://www.engadget.com/2016/03/19/reliable-stagefright-android-exploit/.

Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet.
Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free.
In *6th Software Security, Protection, and Reverse Engineering Workshop, SSPREW 2016, Los Angeles, CA, USA, December 5-6, 2016*, pages 1–12, 2016.

Josselin Feist, Laurent Mounier, and Marie-Laure Potet.
Using static analysis to detect use-after-free on binary code.
In *1st Symposium on Digital Trust in Auvergne*, 2014.

Josselin Feist, Laurent Mounier, and Marie-Laure Potet.
Guided dynamic symbolic execution using subgraph control-flow information.
In *SEFM*, Lecture Notes in Computer Science, pages 76–81. Springer, 2016.

Josselin Feist, Laurent Mounier, and Marie-Laure Potet.
Statically detecting Use-after-Free on binary code.
2013 JCVHT.

Patrice Godefroid, Michael Y. Levin, and David A. Molnar.
Sage: Whitebox fuzzing for security testing.
*ACM Queue*, 10(1):20, 2012.

# References III


Dan Goodin.
most serious linux privilege-escalation bug ever is under active exploit.
http://arstechnica.com/security/2016/10/most-serious-linux-privilege-escalation-bug-ever-is-under-active-exploit/.

Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos.
Dowsing for overflows: A guided fuzzer to find buffer boundary violations.
In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 49–64, Berkeley, CA, USA, 2013. USENIX Association.

Kangjie Lu, Marie-Therese Walter, David Pfaff, N Stefan, Wenke Lee, and Michael Backes.
Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying.
NDSS, 2017.

Robert McMillan.
How heartbleed broke the internet and why it can happen again.
https://www.wired.com/2014/04/heartbleedslesson/.

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov.
Addresssanitizer: A fast address sanity checker.
In *USENIX ATC 2012*, 2012.

Mark Seaborn and Thomas Dullien.
Exploiting the dram rowhammer bug to gain kernel privileges.
*Black Hat*, 2015.

# References IV


Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.
Driller: Augmenting fuzzing through selective symbolic execution.
In *NDSS*. The Internet Society, 2016.

Valgrind.
Memcheck.
`http://valgrind.org/info/tools.html`.

Wikipedia.
Operation aurora, 2013.
[Online; accessed 01-March-2013].

Cristian Zamfir and George Candea.
Execution synthesis: a technique for automated software debugging.
In Christine Morin and Gilles Muller, editors, *EuroSys*, pages 321–334. ACM, 2010.

## *memLoc*

- The **constructor** determines the region of a *memLoc*;
- 6 regions:
  - **Globals** of *addr*
  - **Registers** of *reg_name*
  - **He** of *chunk* $\times$ *offset* (holding heap elements)
  - $\mathbf{Init_{reg}}$ of *id* $\times$ *offset* (holding initial values of registers)
  - $\mathbf{Init_{mem}}$ of *id* $\times$ *offset* (holding initial values of the memory)
  - $\top_{\mathbf{loc}}$.

## *valueSet*

- The **constructor** determines the base of a *valueSet*;
- A $valueSet = \{base \times \{\mathbb{N}\}\}$, 4 bases:
  - **C***onstant*
  - **H***e* of *chunk*
  - $\mathbf{Init_{reg}}$ of *id*
  - $\mathbf{Init_{mem}}$ of *id*.

# Oracle

## Use-After-Free in a trace

- Use-After-Free detection on a trace is not so easy;
- Trouble with aliases.

```
int *p=malloc();
p_alias=p;
free(p);
if(cond){
 p=malloc()
}
else{
 p=malloc();
 p_alias=p;
}
*p=0; // never uaf
*p_alias=0; // uaf if cond
```

Two paths (according *cond*), yet *p* is always equal to *p_alias*.
Only Use-After-Free in *p_alias* if (*cond*).

# Oracle

## Use-After-Free in a trace

*(i)* $t = (\ldots, n_{alloc}(size_{alloc}), \ldots, n_{free}(a_f), \ldots, n_{use}(a_u))$;

*(ii)* $a_f$ is a reaching definition of the block returned by $n_{alloc}$;

*(iii)* $a_u$ is a reaching definition of an address in the block returned by $n_{alloc}$.

## SMT-based solution

- Could use data dependencies analysis but:
  - Traces incomplete, need to have stubs for data dependencies;
- Data dependencies kept *implicitly* with path predicate.

# Oracle

## SMT-based solution

- New path predicate $\varphi$, *malloc* returns $S_{alloc}$, inputs conc;
- $\Phi' = (a_f \neq S_{alloc}) \vee (a_u \notin [S_{alloc}, S_{alloc} + size_{alloc} - 1]);$
- Oracle: $\varphi \wedge \Phi'$ is UNSAT $\rightarrow$ Use-After-Free.

## Explanation

- Instead of $(\forall X \rightarrow SAT)$ we use $(\exists \overline{X} \rightarrow UNSAT)$;
- $a_f \neq S_{alloc}$: the pointer given as the parameter for `free` is not the one allocated at $n_{alloc}$ (negation of property (ii));
- $a_u \notin [S_{alloc}, S_{alloc} + size_{alloc} - 1]$: the pointer used is not a reaching definition of the pointer allocated at $n_{alloc}$ (negation of property (iii)).

# Oracle

```c
int *p=malloc(4);
p_alias=p;
free(p);
if(cond){
 p=malloc()
}
else{
 p=malloc();
 p_alias=p;
}
*p=0; // never uaf
*p_alias=0; // uaf if cond
```

## First path

- Path predicate:
  $\varphi = (p_0 = S_{alloc} \wedge p\_alias_0 = p_0 \wedge p_1 = 0x8040000)$
- $\Phi' = (p_0 \neq S_{alloc} \vee p\_alias_0 \notin [S_{alloc}, S_{alloc} + 3])$
- $\varphi \wedge \Phi'$ UNSAT: there is a Use-After-Free.

# Oracle

```
int *p=malloc(4);
p_alias=p;
free(p);
if(cond){
 p=malloc()
}
else{
 p=malloc();
 p_alias=p;
}
*p=0; // never uaf
*p_alias=0; // uaf if cond
```

## Second path

- Path predicate: $\varphi = (p_0 = S_{alloc} \wedge p\_alias_0 = p_0 \wedge p_1 = 0x8040000) \wedge p\_alias_1 = p_1)$
- $\Phi' = (p_1 \neq S_{alloc} \vee p\_alias_1 \notin [S_{alloc}, S_{alloc} + 3])$
- $\varphi \wedge \Phi'$ SAT (e.g., $S_{alloc} = 0$), no Use-After-Free.

# Initial Memory

```c
char key[]="ABC";

int f(int i){ // 0 <= i < 3
        if(key[i] == 'B'){
                // dest to reach
        }
}
```

$$\Phi_{t_4} \triangleq 0 \leq i < 3 \wedge key[i] \neq' B'.$$

The inversion of the last condition leads then to:

$$\Phi'_{t_4} \triangleq 0 \leq i < 3 \wedge key[i] =' B'.$$

A solution: $i = 2; key[2] =' B'$. Yet $key[2]$ is not user-controllable.

# Initial Memory

## Possible solutions

- Let a free (symbolic) uninitialized memory $\rightarrow$ lost of the correctness;
- Initial state: a snapshot $\rightarrow$ heavy for the solver;
- User define the initial state $\rightarrow$ complex use;
- If all read are concretized $\rightarrow$ concretizing every byte read before being written

# Initial Memory

## Our solution

- (i) We consider as *valid* only models with constraints on symbolic variables corresponding to inputs;
- (ii) We refine the path predicate if it generates an invalid model;
- Refine = re-executing the trace and gathering concrete values;
- Recursively until a valid model is found (or *UNSAT*);
- Necessary for `JasPer`,
- Possible improvements (gathering several values in one round, ...).

# Libraries

## Handling libraries

- Libraries are widely present in real-world;
- Do not want to explore all of them;
- We use two solutions:
  - Stubs $\rightarrow$ model effects on path predicate without tracing instructions (e.g.: if *realloc* returns new pointer, it performs copy of data);
  - Library Driven Heuristics (LDH) $\rightarrow$ known-behavior used to improve guiding.

# Library Driven Heuristics (LDH): Example

```
p=malloc(size) ;
if(p == NULL)
{
 // path to trigger
}
```

## Example

- To trigger the path → *malloc* needs to returns 0.

## *malloc* heuristic

- Principle: on allocation functions, try as parameter a large number.

# Library Driven Heuristics (LDH): Example

```
read(f,tmp,255);
for(i=0;i<255;i++){
  if(tmp[i]=='\0') break;
  buf[i] = tmp[i];
)
buf[i]='\0';
if(strcmp(buf,"this is really bad") == 0)
  ..
```

## Example

- Comparison on string whose length depends of loop iteration number;
- Here, every time loop iterates: $buf[i] \neq$ '\0';
- To solve *strcmp* condition, need a trace unrolling 18 times the loop.

# Library Driven Heuristics (LDH): Example

```
read(f,tmp,255);
for(i=0;i<255;i++){
  if(tmp[i]=='\0') break;
  buf[i] = tmp[i];
)
buf[i]='\0';
if(strcmp(buf,"this is really bad") == 0)
  ..
```

## *strcmp* heuristic

- Principle: on this pattern, use size of constant strings passed to `strcmp` to find the desired iteration.

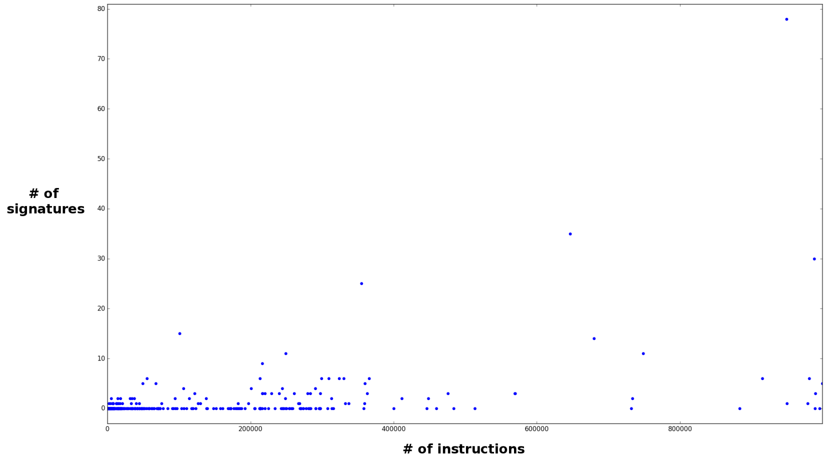
# CVE on JasPer

```
MIF
component
```

Figure: PoC of CVE-2015-5221 (test.plain)

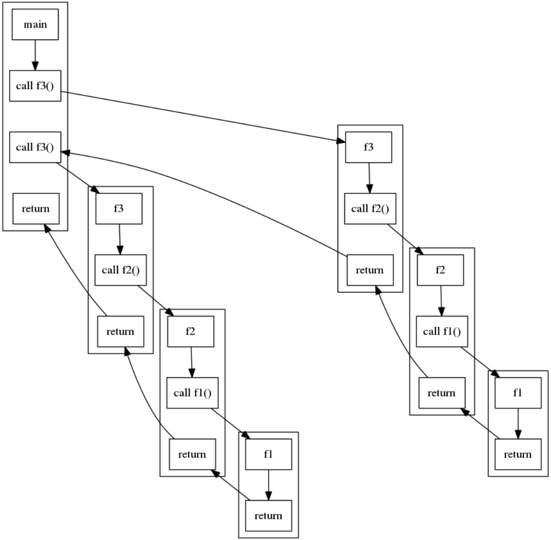
```
jasper --input test.plain --input-format mif --output out --output-format mif
```

# Static Analysis Results

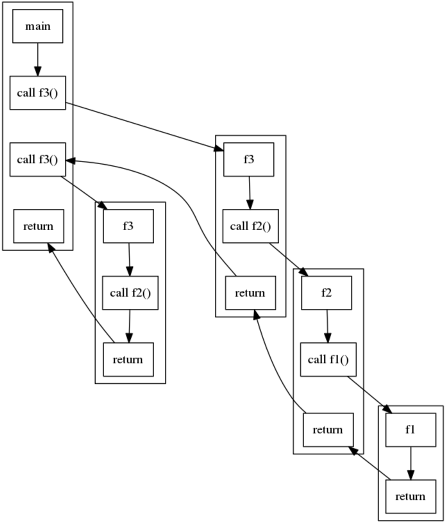


# Inlining

```
void f1(){
 return ;
}

void f2(){
    f1();
    return ;
}

void f3(){
    f2();
    return ;
}

void main(){
    f3();
    f3();
}
```



# Inlining bounded by size (4)



# Inlining bounded by depth (2)