Statically Detecting Use After Free on Binary Code

Josselin Feist, Laurent Mounier, Marie-Laure Potet
firstName.surname@imag.fr

VERIMAG
University of Grenoble

GreHack - November 2013

Plan

whoami

Feist Josselin
Phd student @ Verimag (French lab)
Under the supervision of Marie-Laure Potet & Laurent Mounier

Context

Vulnerability analysis

- Automatize as much as possible the vulnerability detection step by static analysis
- From a vulnerability, formalization of its exploitability

Scientific challenges

- Recent vulnerabilities and yet understudied
- Static analysis at the binary level (scalability/accuracy)
- Memory models adapted for exploitability and symbolic analyses

Our approach/Objective

Static and dynamic analysis

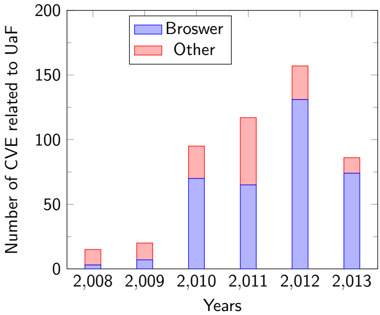
- Using static analysis in order to slice interesting behaviours
  - structural patterns and static taint analysis ([RM12])
- Using static/dynamic analysis for exploitability condition
  - Symbolic exploitability conditions and memory model ([GMPR13])

Plan

Motivations

Motivations

- *Use-After-Free* more and more frequent
- Some dynamic tools (Fuzzing + custom alloc [SBPV12] / Undangle [CGMN12])
- BugWise [Ces13]



Definitions

Dangling pointer

A dangling pointer is a pointer that points to a memory area that has been freed, or reallocated by another pointer.

```
int *p;
p=(int*)malloc(...);
free(p);
```

Use-After-Free

Use-After-Free corresponds to for the use of a dangling pointer.

```
int *p;
p=(int*)malloc(...);
free(p);
[....];
*p=0;
```

Dangerousness

- Reallocation of memory area used by p ?

UaF detection and exploitability

Specificity of UaF

- No easy "pattern" (like for buffer overflow / string format)
- Trigger of several dispatched events (alloc/free/use)
- Strongly depends on the allocation/liberation strategy

Difficulties

- Have the right level of heap abstraction / memory model
- Possibility to describe custom allocator
- Analyze programs with real size
  $\rightarrow$ scaling approach

Plan

Proposed approach

Approach

- 2 steps :
  - 1 : Detection of *Use-After-Free*
    - Value analysis
    - Characterization of *Use-After-Free*
  - 2 : Study of exploitability
    - Ongoing works
- Goal : extract subgraphs of CFG leading to *Use-After-Free* $\rightarrow$ perform the studie of exploitability on specific part of the program
- Input : Heap functions (malloc/free/wrapper...)

Outline

Detection : value analysis

Detection

- Track the use of pointers
- Know the size of allocations
- One allocation = **new** memory area

Value analysis

- Abstract Interpretation
- VSA from [BR04]
- Lighter version
  - Loop without unrolling (allocation-site abstraction)
  - Less accurate (security not safety)
  - Pointer Arithmetic → no track
- VSA → Track of alias

Memory model

Modeling heap

- HE = all possible memory blocks in the heap
- Member of HE represented $(base, size)$ (simplified in *chunk*)
- $HA(pc)$ (resp. $HF(pc)$) member of HE allocated (resp. freed)
- $HA : PC \rightarrow \mathcal{P}(HE)$
- $HF : PC \rightarrow \mathcal{P}(HE)$

Outline

VSA : example

```
1  typedef struct {
2   void (*f)(void);
3  } st;
4
5  int main(int argc, char * argv[])
6  {
7   st *p1;
8   char *p2;
9   p1=(st*)malloc(sizeof(st));
10  free(p1);
11  p2=malloc(sizeof(int));
12  strcpy(p2,argv[1]);
13  p1->f();
14  return 0;
15 }
```

Code	VSA	Heap
`9 : p1=(st*)malloc(sizeof(st))`	AbsEnv = (((Init(EBP), -4),(⊥)), ((Init(EBP), -8),(⊥)))	$HA = \emptyset$ $HF = \emptyset$
`10 : free(p1)`	AbsEnv = (((Init(EBP), -4),(⊥)), ((Init(EBP), -8),(⊥)))	$HA = \emptyset$ $HF = \emptyset$
`11 : p2=malloc(sizeof(int))`	AbsEnv = (((Init(EBP), -4),(⊥)), ((Init(EBP), -8),(⊥)))	$HA = \emptyset$ $HF = \emptyset$

VSA : example

```
1  typedef struct {
2   void (*f)(void);
3  } st;
4
5  int main(int argc, char * argv[])
6  {
7   st *p1;
8   char *p2;
9   p1=(st*)malloc(sizeof(st));
10  free(p1);
11  p2=malloc(sizeof(int));
12  strcpy(p2,argv[1]);
13  p1->f();
14  return 0;
15 }
```

Code	**VSA**	**Heap**
9 : `p1=(st*)malloc(sizeof(st))`	AbsEnv = (((Init(EBP), -4),($chunk_0$)), ((Init(EBP), -8),($\perp$)))	$HA = \{chunk_0\}$ $HF = \emptyset$
10 : `free(p1)`	AbsEnv = (((Init(EBP), -4),($\perp$)), ((Init(EBP), -8),($\perp$)))	$HA = \emptyset$ $HF = \emptyset$
11 : `p2=malloc(sizeof(int))`	AbsEnv = (((Init(EBP), -4),($\perp$)), ((Init(EBP), -8),($\perp$)))	$HA = \emptyset$ $HF = \emptyset$

VSA : example

```c
1  typedef struct {
2   void (*f)(void);
3  } st;
4
5  int main(int argc, char * argv[])
6  {
7   st *p1;
8   char *p2;
9   p1=(st*)malloc(sizeof(st));
10  free(p1);
11  p2=malloc(sizeof(int));
12  strcpy(p2,argv[1]);
13  p1->f();
14  return 0;
15 }
```

Code	VSA	Heap
`9 : p1=(st*)malloc(sizeof(st))`	AbsEnv = (((Init(EBP), -4),($chunk_0$)), ((Init(EBP), -8),($\perp$)))	$HA = \{chunk_0\}$ $HF = \emptyset$
`10 : free(p1)`	AbsEnv = (((Init(EBP), -4),($chunk_0$)), ((Init(EBP), -8),($\perp$)))	$HA = \emptyset$ $HF = \{chunk_0\}$
`11 : p2=malloc(sizeof(int))`	AbsEnv = (((Init(EBP), -4),($\perp$)), ((Init(EBP), -8),($\perp$)))	$HA = \emptyset$ $HF = \emptyset$

VSA : example

```
1  typedef struct {
2   void (*f)(void);
3  } st;
4
5  int main(int argc, char * argv[])
6  {
7   st *p1;
8   char *p2;
9   p1=(st*)malloc(sizeof(st));
10  free(p1);
11  p2=malloc(sizeof(int));
12  strcpy(p2,argv[1]);
13  p1->f();
14  return 0;
15 }
```

Code	VSA	Heap
`9 : p1=(st*)malloc(sizeof(st))`	AbsEnv = (((Init(EBP), -4),($chunk_0$)), ((Init(EBP), -8),($\bot$)))	$HA = \{chunk_0\}$ $HF = \emptyset$
`10 : free(p1)`	AbsEnv = (((Init(EBP), -4),($chunk_0$)), ((Init(EBP), -8),($\bot$)))	$HA = \emptyset$ $HF = \{chunk_0\}$
`11 : p2=malloc(sizeof(int))`	AbsEnv = (((Init(EBP), -4),($chunk_0$)), ((Init(EBP), -8),($chunk_1$)))	$HA = \{chunk_1\}$ $HF = \{chunk_0\}$

Transfer Function

Abstraction of heap functions

$f_{malloc}(pc, HA, HF, AbsEnv, ad, id\_max) = (HA', HF', r, id\_max')$
where:

$$
\begin{aligned}
r &= (base_{id\_max}, size(AbsEnv(ad))) \\
HA' &= HA \leftarrow \{pc \mapsto (HA(pc) \cup \{r\})\} \\
HF' &= HF \\
id\_max' &= id\_max + 1
\end{aligned}
$$

$f_{free}(pc, HA, HF, AbsEnv, ad) = (HA', HF')$
where:

$$HF' = HF \leftarrow \{pc \mapsto (HF(pc) \cup (AbsEnv(ad) \cap HE))\}$$
$$HA' = HA \leftarrow \{pc \mapsto (HA(pc) \setminus (AbsEnv(ad) \cap HE))\}$$

Outline

Detection: characterization of *Use-After-Free*

AccessHeap

AccessHeap returns all elements of *HE* that are *accessed* at *pc*

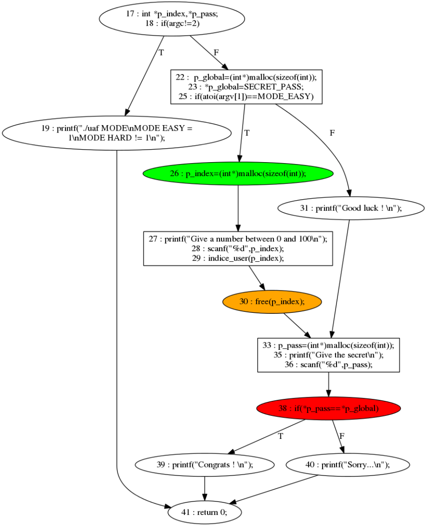
For example :

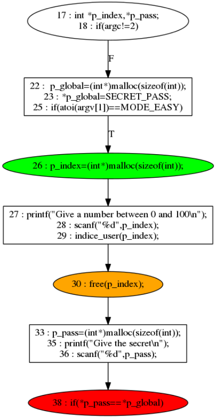
- $AccessHeap(LDM\ ad,, reg) = AbsEnv(ad) \cap HE.$
- $AccessHeap(STM\ reg,, ad) = AbsEnv(ad) \cap HE$

Research the use of a freed element of the heap

- $EnsUaf = \{(pc, chunk) \mid chunk \in AccessHeap(pc) \cap HF(pc)\}$
- Extraction of executions leading to each *Use-After-Free*
  - $pc_{entry} \rightarrow pc_{alloc}$
  - $pc_{alloc} \rightarrow pc_{free}$
  - $pc_{free} \rightarrow pc_{uaf}$

Detection : example





Outline

Implementation

Characteristic

- *IDA Pro* (graph recovery) + *BinNavi* (framework to convert into REIL [Zyn] and VSA)
- *Jython* $\simeq$ 3000 lines

VSA

- Loops are unrolled 0 and 1 times
- *Naive* version of inter-procedural (inlinning)

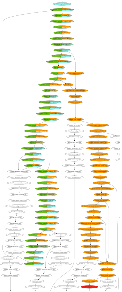
Validation

- Validation of the approach on simple examples
- Evaluation on real CVE

Relevance of the approach

Real *Use-After-Free*

- ProFTPD : CVE 2011-4130, studied by Vupen ([Vup])
- VSA was able to follow struct, function pointer, global var...
- Assisted detection (subset of 10 functions).
- From 2200 nodes → 460, 30 min on i7-2670QM



Discussions on the approach

Separating detection / exploitability

Detection : one allocation = new chunk

Exploitability : one allocation = new or freed chunk

- Triggering *Use-After-Free* independent of the allocation strategy
  - Programming error, always present
  - "Cause" of *Use-After-Free*
- Exploitability of *Use-After-Free* depending on the allocation strategy
  - What has happened between the free / use of the item?
  - " Consequence" of *Use-After-Free*
- Advantage of this approach:
  - Using "classic" technics for detection
  - Study of exploitability on a subset of possible executions of the program

Plan

1. Introduction
2. Use-After-Free
3. Proposed approach
   - Value Analysis
   - VSA example
   - Detection
   - Implementation
4. Exploitability & Conclusion
5. Bibliography

Exploitability : ongoing works

Steps

We consider a Uaf as exploitable if another pointer point to the same memory zone ($\sim$ alias unwanted).

1. Determine paths where new allocations take place between the free and use locations
2. Determine if some allocations can reallocate the same memory area: based on a particular allocation strategy (worst case, all allocations are considered as dangerous)
3. Is the size of new allocations a tainted value? Is the content modified by a tainted value?
4. How is the AccessHeap used: a read, write or jump patterns?

Exploitability : ongoing works

Reallocate of the same memory area

- Simulate an allocator on each "heap operation path" replaying VSA
- Allocator modelisation (with potentially a new heap model):
  - Define some general behaviour/property of allocator :
    - → P1 : Heap space is divided into blocks. Blocks are classified according to their size and state (allocated/freed)
    - → P2 : A new block can take place into a freed block
    - → P3 : A freed block can be split
    - → P4 : Two freed blocks can be consolidated
    - → ...

Conclusions

Conclusions

- PoC of a graph slicing with the aim to help UaF studies
- Ideas to go further (studies of exploitability)
- Uaf difficult to find with dynamic analysis, static can help !

Perspectives

Perspectives

- Use of subgraphs and VSA for smart fuzzing / symbolic execution
- More efficient implementation
  - Refine VSA
  - Change IR ? (Binsec → DBA [BHL+11], more accurate CGF..)
  - Change language (Jython very slow..)
- Detection of home-made allocators (MemBrush [CSB13])
- Complexity of *Use-After-Free* in Internet broswer (several allocation locations including GC, heap spraying)

Plan

Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent.
The bincoa framework for binary code analysis.
In *Proceedings of CAV'11*, pages 165–170, Berlin, Heidelberg, 2011. Springer-Verlag.

Gogul Balakrishnan and Thomas W. Reps.
Analyzing memory accesses in x86 executables.
In Evelyn Duesterwald, editor, *CC*, volume 2985 of *LNCS*, pages 5–23. Springer, 2004.

Silvio Cesare.
Bugalyze.com - detecting bugs using decompilation and data flow analysis.
In *BlackHatUSA*, 2013.

Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa.
Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities.
In Mats Per Erik Heimdahl and Zhendong Su, editors, *ISSTA*, pages 133–143. ACM, 2012.

Xi Chen, Asia Slowinska, and Herbert Bos.
Who allocated my memory? detecting custom memory allocators in c binaries.
In WCRE13, 2013.

Gustavo Grieco, Laurent Mounier, Marie-Laure Potet, and Sanjay Rawat.
A stack model for symbolic buffer overflow exploitability analysis (extended abstract).
In *5th Workshop on the Constraints in Software Testing, Verification and Analysis CSTVA 2013 (in association with ICST 2013)*. IEEE, 2013.

Sanjay Rawat and Laurent Mounier.
Finding buffer overflow inducing loops in binary executables.
In *Proceedings of Sixth International Conference on Software Security and Reliability (SERE)*, pages 177–186, Gaithersburg, Maryland, USA, 2012. IEEE.

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

Vupen.
Technical analysis of proftpd response pool use-after-free (cve-2011-4130).
http://www.vupen.com/blog/20120110.Technical_Analysis_of_ProFTPD_Remote_Use_after_free_CVE-2011-4130_Part_I.php.

Zynamics.
Reil language specification.
http://www.zynamics.com/binnavi/manual/html/reil_language.htm.