# Static analysis for exploitable vulnerability detection

Marie-Laure Potet

**VERIMAG**
University of Grenoble

September 2014

# Outline

# Outline

# The present

*"A software flaw that may become a security threat . . . "*

invalid memory access (e.g., buffer overflows, dangling pointers), arithmetic overflow, race conditions, etc.

- Still present in current applications and OS kernels:
  5000 in 2011, 5200 in 2012, **6700 in 2013** . . . [Symantec]
- Multiple consequences:
  program crash, malware injection, priviledge escalation, etc.

## A business

A market has been established for vulnerabilities
Companies, governments and criminals buy vulnerability information and accompanying exploits
Up to $250,000 for a single zero - day exploit

# Practice in terms of vulnerability analysis

1. Identification of flaws
   - dangerous patterns, fuzzing and crashes identification ...
2. Possibility of exploit (exploitability)
   - poc elaboration, taint analysis, crash analysis ...
3. Building an real exploit
   - hijacking countermeasures (sandboxing, DEP, ASLR) using well-established techniques and forms of shellcodes

Current practice : fuzzing + manual crash analysis

⇒ Challenges : classification of flaws that are exploitable, false positive/negative, real exploits (dedicated expertise)

# Example 1

```c
void bufCopy(char *dst, char *src)
{
        char *p = dst;
        while (*src != ’\0’) *p++ = *src++;
        *p = ’\0’;
}
```

```c
void CallbufCopy(char *src)
   {char dst[4] ;
    bufCopy(dst, src);
}
```

1. Flaw: buffer overflow if no 0 in the first four characters
2. Poc : control flow hijacking if the return address is erased
3. Weaponized exploit : DEP ($\rightarrow$ ROP), ASLR ($\rightarrow$ address leaking, unrandomized library . . . ) Sandboxing ($\rightarrow$ own vulnerability)

# Outline

# Used Static analysis technics

Static analysis : all traces can be taken into account (or a significant part of), possibility of symbolic reasoning

Technics we use:

- Taint and dependency analysis
  - impact of inputs, data and control dependencies
- Value analysis
  - Determine set of values including reachable values (abstract interpretation)
- Symbolic execution (or concolic)
  - Build path predicates and resolve them by SMT solvers.
    Example 1 with size(dst)=4 and size(src)=8:

```
p0 = dst0
and not(*src0='\0') and *p0=*src0 and p1=p0+1 and src1=src0+1
and not(*src1='\0') and *p1=*src1 and p2=p1+1 and src2=src1+1
and *src2='\0'and *p2='\0'
```

    Pathcrawler/Klee: 9 test cases (4+4+1)

# Static analysis and vulnerability detection

Applications for vulnerability detection:

- identification of sensible parts of code (sophisticated patterns involving values)
- input generation from symbolic paths (slicing)
- generalization of traces (exploitability)

$\Rightarrow$ Exploitability only makes sense at the binary level

Challenges :

- Taint and dependency analysis require a value analysis
- bitvector representation and adapted memory models
- scalability/completeness

# Binary level and dependency

⇒ Taint analysis at the source level:

```
int x, *p, y;
x = 3 ;
p = &x ;
y = *p + 4 ;                    -- y is untainted
```

⇒ Taint analysis at the assembly level:

| Assembly | Value analysis result |
|---|---|
| /* x=3; */ | |
| mov [ebp-4], 3 | Mem[ebp-4]=3 |
| lea eax, [ebp-4] | eax = ebp-4 |
| /* p = &x ;*/ | |
| mov [ebp-8], eax | Mem[ebp-8] = ebp-4 |
| mov eax, [ebp-8] | eax = Mem[ebp-8] |
| /* y = *p+4 ; */ | |
| mov eax, [eax] | eax = Mem[Mem[ebp-8]] = Mem[ebp-4] |
| add eax, 4 | eax = Mem[ebp-4] + 4 |
| mov [ebp-12], eax | Mem[ebp-12] = eax = Mem[ebp-4] + 4 = 3 + 4 |

Mem[ebp-12] is untainted.

# Adapted memory models

Verification:

- detection of undefined behaviors
- separate regions (stack frames, block allocation, array ...)

Vulnerability detection:

- exploitation of undefined behaviors
- memory layout representation (flat memory)

Problems:

- value analysis : weak update/ strong update
- Symbolic reasoning :

$$select(store(t, i, v), i) = v$$
$$select(store(t, i, v), j) = select(t, j, v) \text{ } if \text{ } i \neq j$$

# Exploitability

⇒ Generalization of a crash adding constraints (PC corruption, writing a determined portion of memory ...).Example (12 loop traversals for rewriting the return address):

```
p0 = dst0                                                     // initializati
and not (*src0='\0') and *p0=*src0 and p1=p0+1 and src1=src0+1  // ex. 1
...    ...
and not(*src8='\0')  and *p8=*src8   and p9=p8+1   and src9=src8+1  // ex. 9
and not(*src9='\0')  and *p9=*src9   and p10=p9+1  and src10=src9+1  // ex. 10
and not(*src10='\0') and *p10=*src10 and p11=p10+1 and src11=src10+1  // ex. 11
and not(*src11='\0') and *p11=*src11 and p12=p11+1 and src12=src11+1  // ex. 13
and *src8='A'and *src9='B'and *src10='C' and *src11='D' // \@ payload
and *src12='\0' and *p12='\0'
```

AEG a new domain (Sean Heelan, David Brumley,BinSec).
Challenges:

- how to generalize?
- memory models between flat models and fine-grained regions
- exploitability conditions for other vulnerabilities

# Our approach

⇒ Identifying exploitable paths and building appropriate inputs

- Using static analysis in order to slice interesting behaviours
  - structural patterns and static taint analysis
- Using static/dynamic analysis for exploitability condition
  - Symbolic exploitability conditions and dependency
- Using concolic or genetic approach to produce inputs
  - guided fuzzing

⇒ Buffer overflow : SERE11 (BO pattern), SAW'14 (inter-procedural static taint analysis), ECND10, SECTEST11 (fitness functions and mutations)

⇒ Prototype: IdaPro+REIL

# Outline

# Outline

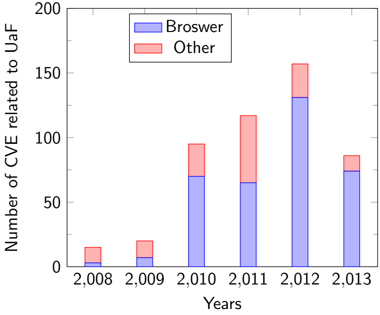
# Use after free : dangling pointer + access

```

typedef struct {
 void (*f)(void);
} st;

void nothing()
{
 printf("Nothing\n");
}

int main(int argc, char * argv[])
{
 st *p1;
 char *p2;
 p1=(st*)malloc(sizeof(st));
 p1->f=&nothing;
 free(p1);                        // p1 freed
 p2=malloc(strlen(argv[1]));      // possible re-allocation
 strcpy(p2,argv[1]);
 p1->f();                         // Use
 return 0;
}
```

# Motivations

## Motivations

- *Use-After-Free* more and more frequent (CVE-2014-0322 (internet explorer), CVE-2014-1512 (firefox,thunderbird))
- Static approach for finding exploitable vulnerabilities
  $\rightarrow$ an adapted modelling of the heap




https://web.nvd.nist.gov/view/vuln/search, 4 june 2013

# State of art

## 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
- source level detection tools

## Binary code

On binary code, state of the art focused more on dynamic analysis

- Fuzzing + custom allocator (AddressSanitizer)
- Exploit studied after UaF found (Undangle)
- New Microsoft protections for navigators (separated heaps, safe memory management) (June 2014)

# Proposed approach

Goal : extract subgraphs of CFG leading to exploitable *Use-After-Free*

## Approach

- 2 steps :
  - Step 1 : Detection of *Use-After-Free*
    - Value analysis
    - Characterization of *Use-After-Free*
  - Step 2 : Exploitability of *Use-After-Free*
    - Determining possible re-allocations
    - Exploitability condition (ongoing work)

Semi-automatic : choice of allocation strategy properties

# Outline

# Memory model and VSA

## Modelling heap

- $HE$ = all possible memory blocks in the heap
- Member of $HE$ represented $(heap_i, size_i)$ (simplified in $chunk_i$)
- $HA(pc)$ (resp. $HF(pc)$) member of $HE$ allocated (resp. freed)
- $HA : PC \to \mathcal{P}(HE)$
- $HF : PC \to \mathcal{P}(HE)$
- $HA(pc) \cap HF(pc) = \emptyset$

## VSA for detection

- Track allocation, free and heap accesses
- size of allocation (for exploitability)
- One allocation = new *chunk*

# Transfer functions for heap operations

**function** malloc($pc, size$)
$id := id\_max$;
$id\_max++$;
$HA := HA \leftarrow \{pc \mapsto (HA(pc) \cup \{(base_{id}, size)\})\}$;
$point\_alloc := point\_alloc \leftarrow \{(base_{id}, size) \mapsto pc\}$;
**return** $(base_{id}, size)$
**end function**

**function** Free($pc, (base_x, size)$)
$HA := HA \leftarrow \{pc \mapsto (HA(pc) \setminus \{(base_x, size)\})\}$;
$HF := HF \leftarrow \{pc \mapsto (HF(pc) \cup \{(base_x, size)\})\}$;
$point\_free := point\_free \leftarrow \{(base_x, size) \mapsto$
$\{point\_free(base_x, size) \cup pc\}\}$;
**end function**

```
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 | AbsEnv | Heap |
|---|---|---|
| `9 : p1=(st*)malloc(sizeof(st))` | **(Init(EBP), -4)** $\mapsto$ $\{\mathbf{chunk_0}\}, \ldots$ | **HA** $= \{\mathbf{chunk_0}\}$<br>$HF = \emptyset$ |
| `10 : free(p1)` | (Init(EBP), -4) $\mapsto$ $\{chunk_0\}, \ldots$ | $HA = \emptyset$<br>**HF** $= \{\mathbf{chunk_0}\}$ |
| `11 : p2=malloc(sizeof(int))` | (Init(EBP), -4) $\mapsto$ $\{chunk_0\}$,<br>**(Init(EBP), -8)** $\mapsto$ $\{\mathbf{chunk_1}\}$ | **HA** $= \{\mathbf{chunk_1}\}, \ldots$<br>$HF = \{chunk_0\}$ |

# Detection: characterization of *Use-After-Free*

## AccessHeap

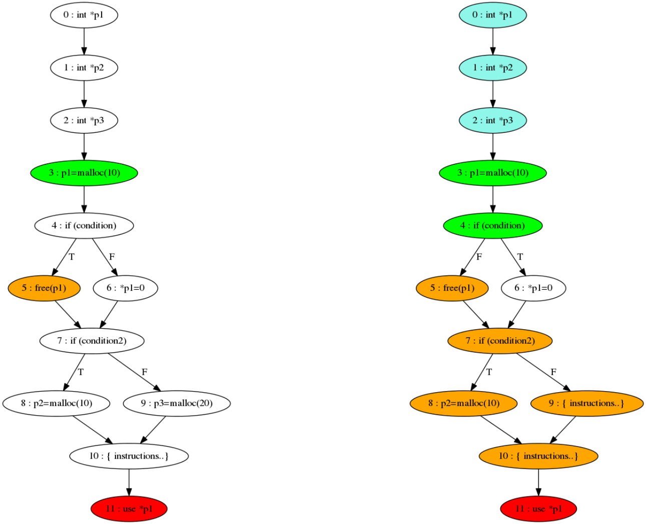
AccessHeap returns all elements of *HE* that are *accessed* at *pc*
Examples with REIL memory transfer instructions:

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

## 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*: all reachable nodes including the following paths:
  - $pc_{entry} \rightarrow pc_{alloc}$
  - $pc_{alloc} \rightarrow pc_{free}$
  - $pc_{free} \rightarrow pc_{uaf}$

# Example: *Use-After-Free* detection and extraction



# Outline

# Exploitability

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

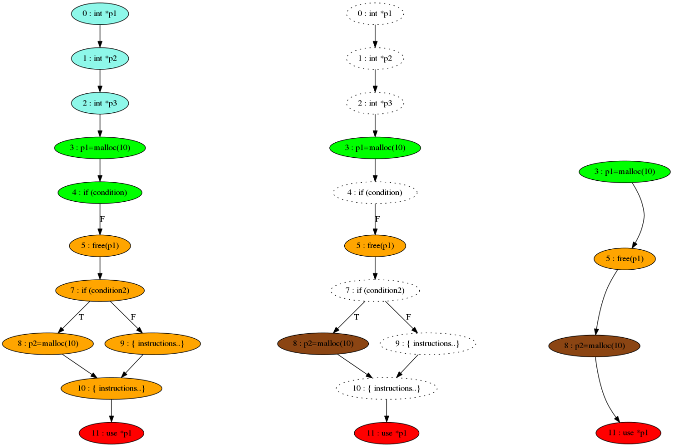
## Steps

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?

# 1. Extracting paths with re-allocations

## Replay allocations between free → use

- Allocation order is important for exploitability
- Find all "heap operations paths" (with loop summary)



## 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
    - → ...

| **Code** | **Heap** |
|---|---|
| `9 : p1=(st*)malloc(sizeof(st))` | $\mathbf{HA} = \{(\mathbf{heap_0}, \mathbf{4})\}$ <br> $HF = <>$ |
| `10 : free(p1)` | $HA = \emptyset$ <br> $\mathbf{HF} = < (\mathbf{heap_0}, \mathbf{4}) >$ |
| `11 : p2=malloc(sizeof(int))` | $\mathbf{HA} = \{(\mathbf{heap_0}, \mathbf{4})\}$ <br> $HF = <>$ |

# 3 and 4. Dangerousity: taintness and type of HeapAccess

```c

typedef struct {
 void (*f)(void);
} st;

void nothing()
{
 printf("Nothing\n");
}

int main(int argc, char * argv[])
{
 st *p1;
 char *p2;
 p1=(st*)malloc(sizeof(st));
 p1->f=&nothing;
 free(p1);
 p2=malloc(strlen(argv[1]));     // size is tainted
 strcpy(p2,argv[1]);             // content of p2 is tainted
 p1->f();                        // Access as a jump
 return 0;
}
```

# Discussions on the approach

## Separating detection / exploitability

- 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" technique for detecting
  - Study of exploitability on a subset of possible executions of the program
  - For an *Use-After-Free* detected opportunity to study several allocation strategies (or worst case)

# Outline

# Implementation

⇒ *Use-After-Free* detection step

## Characteristic

- *IDA Pro* + *BinNavi*
- *Ocaml*

## VSA

- loops are unrolled n times (to be instanciated)
- inter-procedural by inlining
- parametrable memory model (stack frame)

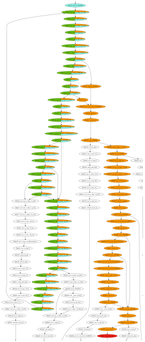
## Validation

- Validation of the approach on simple examples
- Further study of a CVE

# Relevance of the approach

## Real *Use-After-Free*

- ProFTPD : CVE 2011-4130, studied by Vupen
- Structures, function pointer, global variables...
- Assisted detection (subset of 10 functions).
- From 2200 nodes → 460



# Ongoing works

- Use of subgraphs and VSA for smart fuzzing
- An adapted IR and flow graph construction and memory model ANR project (BinSec)
- Exploitability steps (including impact of exploitability)
- Build traces using symbolic exploitability conditions (and allocation strategy)
- Detection of custom allocators
- Complexity of *Use-After-Free* in navigators (several allocation locations including GC, heap spraying)

# Outline

# Outline

# The BinSec project

ANR 2013-2017
CEA-LIST, EADS IW, INRIA Rennes, LORIA, Vupen Security

- Engineering of vulnerability analysis
  - Automatize as much as possible the vulnerability detection step
  - Formalisation of skills in term of exploitability
- Scientific challenges
  - New vulnerabilities such as Use after Free
  - Static analysis at the binary level (scalability/accuracy)
  - Memory models for exploitability and symbolic analyses
  - Representation of self-modifying code

$\Rightarrow$ An IR: DBA

$\Rightarrow$ An open flat-form with CFG recovery a set of basic analysis

# Another application domain

⇒ Smart card applications: injections of fault impacting the code logic (data and control flow)

- Multi-fault
- Embedding fault injection by code mutation
- Use of symbolic execution to evaluate the robustness of code
- Scalability for Binary level (dependency)
- Dependency on memory states

⇒ Lazart: an implementation acting on LLVM IR (ICST'14)

Louis Dureuil'thesis, A starting Project

# Involded People

- Louis Dureuil (Doctorant CEA-Vérimag)
- Josselin Feist (Doctorant Vérimag)
- Roland Groz (LIG, Prof. Grenoble INP)
- Laurent Mounier (MC Université Joseph Fourier)
- Marie-Laure Potet (Prof. Grenoble INP)
- Maxime Puys (Doctorant Vérimag-INRIA)
- Sanjay Rawat (International Institute of Information Technology, Hyderabad, India)

# Outline

# References I


Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier.
A taint based approach for smart fuzzing.
In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Proceedings of SecTest*, pages 818–825, 2012.

Josselin Feist, Laurent Mounier, and Marie-Laure Potet.
Statically detecting use-after-free on binary code.
*Journal of Computer Virology and Hacking Techniques*, online article, January 2014.

Gustavo Grieco, Laurent Mounier, Marie-Laure Potet, and Sanjay Rawat.
A stack model for symbolic buffer overflow exploitability analysis.
In *Proceedings of CSTVA (ICST Workshop)*, pages 216–217, Luxembourg, march 2013. IEEE.

Guillaume Jeanne.
Génération automatique d'exploits à partir de traces d'erreurs.
MR Grenoble INP, september 2014.

Marie-Laure Potet, Josselin Feist, and Laurent Mounier.
Analyse de code et recherche de vulnérabilités.
*Revue MISC*, hors-série, juin 2014.

Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil.
Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections.
In *IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 213–222, 2014.

Sanjay Rawat, Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet.
Combining static and dynamic analysis for vulnerability detection.
*MDV'10, Modeling and Detecting Vulnerabilities workshop, associated to ICST 2010, IEEE digital Library*, 2010.

# References II


Sanjay Rawat and Laurent Mounier.
Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results.
In *Proc. of The Second International Workshop on Security Testing (SECTEST)*. IEEE, 2011.

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.

Sanjay Rawat, Laurent Mounier, and Marie-Laure Potet.
LiSTT: An investigation into unsound-incomplete yet practical result yielding static taintflow analysis.
In *Proceedings of SAW 2014 (ARES Workshop)*, Fribourg (Switzerland), September 2014. IEEE.