Using static analysis to detect use-after-free on binary code

Josselin Feist Laurent Mounier Marie-Laure Potet

Verimag / University of Grenoble - Alpes

France

SDTA 2014 - Clermont-Ferrand

5 décembre 2014

Software Vulnerability

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

invalid memory access, arithmetic overflow, race conditions, etc.

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

⇒ A major security concern ... (and a business !)

editors, security agencies, defense departments, etc.

Vulnerability detection in practice

A 2-phase approach

1. (pseudo-random) fuzzing, fuzzing, . . . and fuzzing
   $\hookrightarrow$ to produce a huge number of **program crashes**
2. in-depth **manual** crash analysis
   $\hookrightarrow$ to identify **exploitable** bugs

Vulnerability detection in practice

A 2-phase approach

1. (pseudo-random) fuzzing, fuzzing, ... and fuzzing
   $\hookrightarrow$ to produce a huge number of **program crashes**
2. in-depth **manual** crash analysis
   $\hookrightarrow$ to identify **exploitable** bugs

Consequences

- A time consuming activity
  (very small ratio "exploitable flaws/simple bugs" !)
- Would require a better **tool assistance** ...

Work objective : automated detection of **Use-after-Free**

Use-after-Free vulnerability (UaF)

- UaF = use of a dangling pointer in a program statement
- occurs in complex memory management applications:
  Firefox (CVE-2014-1512), Chrome (CVE-2014-1713), IE (CVE-2014-1763)

Work objective : automated detection of **Use-after-Free**

Use-after-Free vulnerability (UaF)

- UaF = use of a dangling pointer in a program statement
- occurs in complex memory management applications:
  Firefox (CVE-2014-1512), Chrome (CVE-2014-1713), IE (CVE-2014-1763)

Scientific and technical challenges

- a complex **vulnerability pattern**:
  triggered by # events (alloc, free, use)
- needs a suitable **memory model**:
  heap representation, allocation/de-allocation strategy
- **binary code** analysis:
  low-level representation, scalability/accuracy trade-offs

UaF example 1: information leakage

```
char *login, *passwords;
login=(char *) malloc(...);
[...]
free(login); // login is now a dangling pointer
[...]
passwords=(char *) malloc(...);
   // may re-allocate memory area used by login
[...]
printf("%s\n", login)  // prints the passwords !
```

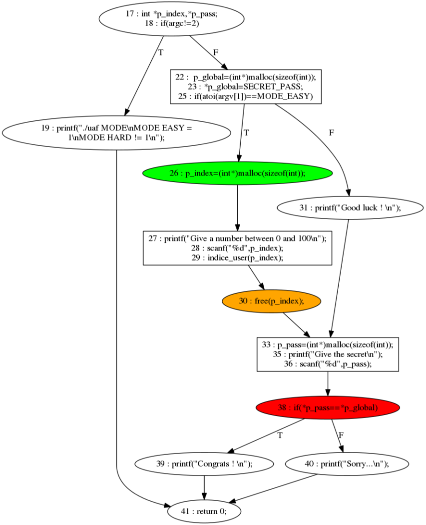
UaF example 2: execution hijacking

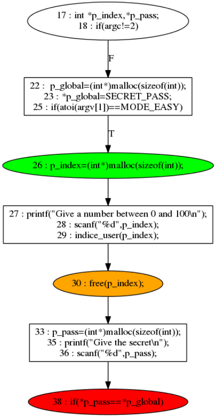
```
typedef struct {
 void (*f)(void);  // pointer to a function
} st;

int main(int argc, char * argv[])
{
 st *p1;
 char *p2;
 p1=(st*)malloc(sizeof(st));
 free(p1); // p1 is now a dangling pointer
 p2=malloc(sizeof(int)); // memory area of p1 ?
 strcpy(p2,argv[1]);
 p1->f();  // calls any function you want ...
 return 0;
}
```

Static detection of UaF (example)

extract Uaf patterns = program slices containing potential UaF





Step 1 : where and how the heap is accessed ?

HE = sets of all heap elements

Compute at each program location l

- the sets of allocated and free heap elements: $HA(l)$, $HF(l)$
  $\rightarrow$ needs to identify **allocation/de-allocation primitives**
- the (abstract) value of each memory location m : $AbsEnv(l, m)$

Step 1 : where and how the heap is accessed ?

HE = sets of all heap elements

Compute at each program location l

- the sets of allocated and free heap elements: $HA(l)$, $HF(l)$
  $\rightarrow$ needs to identify **allocation/de-allocation primitives**
- the (abstract) value of each memory location m : $AbsEnv(l, m)$

Implemented as an abstract interpretation technique

- computes addresses and contents of active memory locations
- tracks heap pointers and pointer aliases
- "lightweight" Value-Set Analysis (VSA)

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),($\perp$)), ((Init(EBP), -8),($\perp$)))	$HA = \emptyset$ $HF = \emptyset$
`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

```
 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

```
 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

```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),($chunk_0$)), ((Init(EBP), -8),($chunk_1$)))	$HA = \{chunk_1\}$ $HF = \{chunk_0\}$

AccessHeap

$$AccessHeap(I) = \text{set of heap elements } \textit{accessed} \text{ at } I$$

Example:

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

Step 2 : slicing the Control-Flow Graph

AccessHeap

$$AccessHeap(l) = \text{set of heap elements } \textit{accessed} \text{ at } l$$

Example:

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

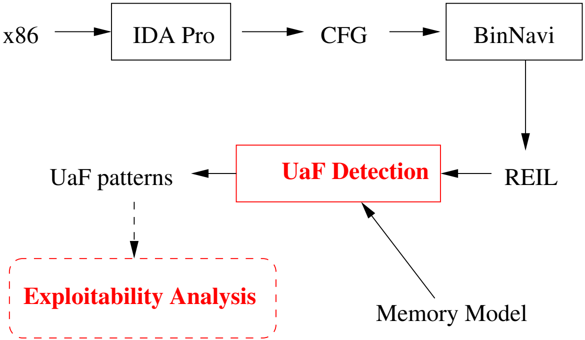
Locate UaF on the CFG

- Use-after-Free = accessing a heap element which is free

  $$EnsUaf = \{(l, h) \mid h \in AccessHeap(l) \cap HF(l)\}$$

- Extraction of executions leading to each *Use-After-Free*
  - $l_{entry} \rightarrow l_{alloc}$
  - $l_{alloc} \rightarrow l_{free}$
  - $l_{free} \rightarrow l_{uaf}$

Implementation



Characteristics

- previous implementation in *Jython*, new one in *OCAML* . . .
- validation with simple examples
- under evaluation on a real CVE (ProFTPD, CVE 2011-4130)

Exploitability Analysis (ongoing work)

$\rightarrow$ decide if an UaF pattern is "likely to be exploitable" ?

Several criteria:

- is e **re-allocated** between "free(e)" and "access(e)" ?
  $\hookrightarrow$ strongly depend on the allocation strategy ...
- is the size and/or content of the allocated element **tainted** ?
  $\hookrightarrow$ dependency analysis from input functions ...
- what **kind** of access(e): read, write, jump ?
- etc.

Proposed approach:

$\Rightarrow$ guided symbolic execution on the UaF pattern ...

Conclusions

- Uaf are difficult to find with pure dynamic analysis ...
  $\rightarrow$ first attempt to statically characterize UaF patterns ?
- PoC tool

Conclusion & Perspectives

Conclusions

- Uaf are difficult to find with pure dynamic analysis . . .
  $\rightarrow$ first attempt to statically characterize UaF patterns ?
- PoC tool

Perspectives

- integrate the tool within the **ANR BinSec plateform**
- detection of home-made allocators ([H. Bos et al])
- real-life *Use-After-Free* in web browsers ?
- exploitability analysis . . .