J Comput Virol Hack Tech (2014) 10:211–217
DOI 10.1007/s11416-014-0203-1

CORRESPONDENCE

Statically detecting use after free on binary code

Josselin Feist · Laurent Mounier · Marie-Laure Potet

Received: 7 October 2013 / Accepted: 11 January 2014 / Published online: 30 January 2014

Abstract We present **GUEB** a static tool detecting Use after Free vulnerabilities on disassembled code. This tool has been evaluated on a real vulnerability in the ProFTPD application (CVE-2011-4130).

1 Introduction

Vulnerability detection classically starts with a first step of fuzzing, exercising applications with potentially dangerous inputs. Traces, produced by fuzzing , are generally classified using some heuristics (for instance traces producing a memory crash). Then, the exploitability of these traces, or similar ones, is manually studied. In order to obtain interesting traces, fuzzers require to identify which parts of an application must be stressed and how (i.e., SQL injection, buffer overflow, etc.). To do that an approach consists in statically identifying vulnerable patterns (from syntactic ones, such as `strcpy` calls, to more sophisticated ones, like in [13]).

We propose here a static approach dedicated to the detection of Use after Free (UaF) patterns in binary code, a vulnerability doubling every year since 2008 [7]. UaF are characterized by the occurrence of two distinct event: the *creation* of a dangling pointer, later followed by an *access* to the memory pointed to by this pointer [1]. Hence, detecting UaF requires to analyze long execution sequences, which is a challenging task when dealing with large applications. Consequently, we believe that a static analysis can provide good results from a scalability point of view, finding UaF patterns that would be hard to detect using pure dynamic approaches. More precisely, our objective is to identify sets of program locations involved in a UaF, providing a first level vulnerability detection step, and applicable to large (binary) codes. Moreover, information collected during this step could also be useful for a subsequent exploitability analysis (i.e., telling if and how some freed memory chunk can be later reallocated or overwritten).

This work was partially funded by the Binsec project (ANR-12-INSE-0002-01).

J. Feist · L. Mounier (✉) · M.-L. Potet
Vérimag Laboratoy, University of Grenoble, Centre Équation,
2 Avenue de Vignate, Gières 38610, France
e-mail: Laurent.Mounier@imag.fr

J. Feist
e-mail: Josselin.Feist@imag.fr

M.-L. Potet
e-mail: Marie-Laure.Potet@imag.fr

1.1 A motivating example

We explain our approach through a motivating example, given on Listing 1[1]. The main idea is that the function `index_user` puts `p`, the function argument, in the global variable `p_global` (line 8) and restores the previous value of `p_global` (just before the end of the function, line 13). But, when the condition `cmp` fails at line 9, `index_user` does not restore the value of `p_global`. This kind of mistake (i.e., forget to restore a previous pointer value), can be found in real programs (this UaF is borrowed from CVE-2011-4130). So, if during the call at line 29 this behaviour happens, `p_global` will point on the same memory area than `p_index` and therefore, after the `free` at line 30, `p_global` will become a dangling pointer. Then, at line 33, `malloc` could return the same memory area[2] as pointed to by `p_index`. In this case, the comparison at line 38 will always be true.

[1] In C for a better understanding, but our analysis operates at the assembly level.

[2] This is the case for instance with the libc.

Listing 1 Motivating example

```
 1  int *p_global;
 2
 3  int cmp() { return *p_global>=MIN && *p_global<=
        MAX; }
 4
 5  void index_user(int *p) {
 6        int *p_global_save;
 7        p_global_save=p_global;
 8        p_global=p;
 9        if(cmp()<=0)    {
10              printf("The secret is greater than
                    50\n");
11              return ; }
12        printf("The secret is less than 50 \n");
13        p_global=p_global_save;  return ;
14  }
15
16  int main(int argc, char * argv[]) {
17        int *p_index,*p_pass;
18        if(argc!=2) {
19              printf("./uaf MODE\nMODE EASY=1\
                   nMODE HARD!=1\n");
20              return 0; }
21
22        p_global=(int*)malloc(sizeof(int));
23        *p_global=SECRET_PASS;
24
25        if(atoi(argv[1])==MODE_EASY) {
26              p_index=(int*)malloc(sizeof(int));

27              printf("Give a number between 0
                   and 100\n");
28              scanf("%d",p_index);
29              index_user(p_index);
30              free(p_index);  }
31        else { printf("Good luck ! \n"); }
32
33        p_pass=(int*)malloc(sizeof(int));
34
35        printf("Give the secret\n");
36        scanf("%d",p_pass);
37
38        if(*p_pass==*p_global)  {
39                    printf("Congrats ! \n"); }
40        else { printf("Sorry...\n"); }
41        return 0; }
```

1.2 Our approach

Although more complex, static analysis provides more complete results than dynamic analysis by allowing to explore the whole set of program executions. For instance, for UaF, dangerous paths have to follow up several events: first, allocate a heap address, second, free this address, and, finally, access to the memory pointed to by this address. Using a dynamic search, we have few chances to find a path fulfilling all these requirements. Furthermore, a static analysis is independent of some execution specificities, such as effective addresses. For instance, in the case of UaF we can detect dangerous behaviours independently of the invoked allocator.

Our contribution, called **GUEB** for Graph of Use-After-Free to Exploit Binary, is based on three steps. First we track heap operations and address transfers, taking into account aliases, using a dedicated value analysis (see Sect. 2.2). Secondly we exploit these results to statically identify UaF vulnerabilities. Finally we extract subgraphs, for each UaF, describing sequentially where the dangling pointer is *created*, *freed* and *used* (see Sect. 3.2). Figure 1 gives the subgraph extracted from Listing 1 (presented at the source level). It identifies bloc 26 (creation of a pointer p), bloc 30 (freeing of this pointer) and bloc 38 (dereferencing of p).

1.3 Some related work

There exists several tools statically tracking UaF in source-level C code (such as Polyspace[3] or Frama-C [9] for instance). These tools are mainly dedicated to safety verification: programs that do not respect some constraints are rejected, such as undefined behaviors in C. In vulnerability analysis we are precisely interested by unpredictable behaviors, that could be successfully exploited. For instance in the case of UaF we are not just interested by the use of a dangling pointer, we want to localise where they are created and how they can be effectively exploited. Getting such detailed information on the memory layout requires to analyze the code at the binary level, taking into account the compiler optimizations and the libraries in use.

However, lifting static analysis to binary codes is quite challenging [3]. On the academic side, there exists several open platforms providing some multi-purpose static analysis (program slicing, CFG construction, data-flow analysis, etc.), and well-defined APIs to access intermediate code representations. One can mention for instance REIL [10], BAP [6], or Bincoa [5]. But none of them provide any on-the-shelf solution for UaF detection.

As a result, when binary code is concerned, to the best of our knowledge, existing UaF detection tools are all based on dynamic approaches. For example, AddressSanitizer [14] requires to compile the binary with a specific library. Undangle [7] is a tool specifically dedicated to UaF analysis: starting from a given execution trace, this tool aims to identify program points where dangling pointers are created, to detect the root of a possible vulnerability. In this sense, it pursues the same objectives as ours. Nevertheless, this approach requires to first identify paths leading to UaF vulnerabilities, to instrument each assembly instruction, which slows down the execution.

The paper is organized as follows: Section 2 describes the memory model and the value analysis used, Sect. 3 presents the method we propose to detect UaF patterns, and Sect. 4 summarizes some experimental results and gives some limitations of the current prototype and directions for future work.

[3] http://www.mathworks.fr/products/polyspace.

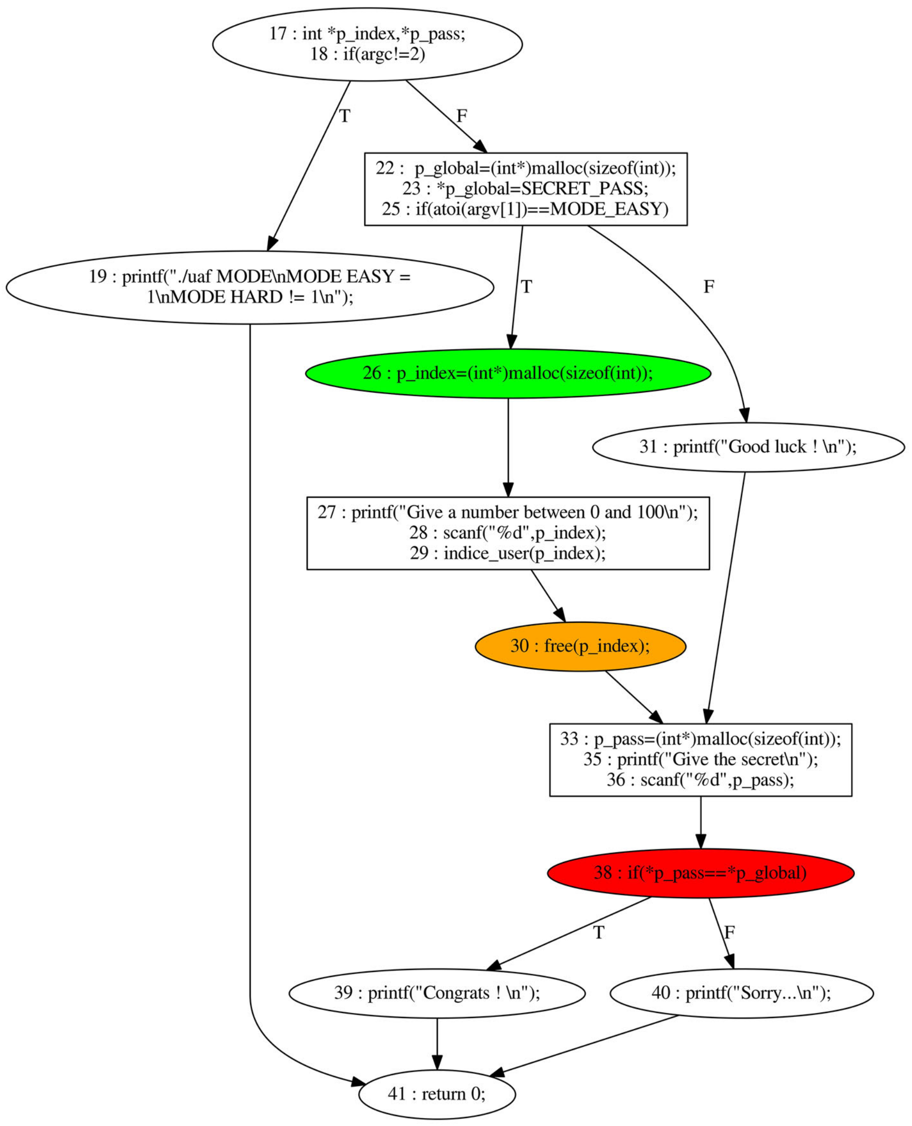

17 : int *p_index,*p_pass;
18 : if(argc!=2)
T
F
22 : p_global=(int*)malloc(sizeof(int));
23 : *p_global=SECRET_PASS;
25 : if(atoi(argv[1])==MODE_EASY)
19 : printf("./uaf MODE\nMODE EASY = 1\nMODE HARD != 1\n");
T
F
26 : p_index=(int*)malloc(sizeof(int));
31 : printf("Good luck ! \n");
27 : printf("Give a number between 0 and 100\n");
28 : scanf("%d",p_index);
29 : indice_user(p_index);
30 : free(p_index);
33 : p_pass=(int*)malloc(sizeof(int));
35 : printf("Give the secret\n");
36 : scanf("%d",p_pass);
38 : if(*p_pass==*p_global)
T
F
39 : printf("Congrats ! \n");
40 : printf("Sorry...\n");
41 : return 0;

Fig. 1 Extracted subgraph for example Listing 1

2 Memory model and value analysis

In this section we detail each of the three steps of the GUEB approach. First, we explain how the stack and heap elements are represented.

2.1 Abstract memory representation

We assume that addresses in the stack are expressed as offsets with respect to the base register EBP. Since inter-procedural analysis is achieved by *procedure in-lining*[4], each stack element is represented by a pair $(EBP_0, offset)$ where EBP_0 is the initial value of EBP. For instance `p_index` is denoted by $(EBP_0, -24)$ and `p_global_save` by $(EBP_0, -36)$. Global variables have constant addresses represented by an identifier, here the variable name (e.g., `p_global`).

Regarding the heap, we define HE as the set of all possible heap elements. An element of HE is a pair $(base, size)$, where $base$ is an allocation identifier and $size$ the allocation size. Such a pair is also denoted as a *chunk*. PC is the set of all program points. We define HA and HF, two functions that respectively associate the set of all currently allocated or freed elements at each point pc ($HA \in PC \to \mathcal{P}(HE)$, $HF \in PC \to \mathcal{P}(HE)$, $\mathcal{P}(S)$ being the power set of S). We use the classical hypothesis (in static pointer verification), telling that each allocation supplies a fresh memory block (although this approach is not realistic to study the exploitability, it is sufficient to detect the vulnerability).

[4] Making our analysis context-sensitive, but not applicable to recursive calls.

Table 1 *VSA* result

Code	AbsEnv	Heap
22 `p_global=(int*)malloc(..)`	$\{(\mathbf{p\_global},(\mathbf{chunk_0}))\}$	$\mathbf{HA} = \{\mathbf{chunk_0}\}$ $HF = \emptyset$
26 `p_index=(int*)malloc(..)`	$\{((\mathbf{EBP_0}, \mathbf{-24}),(\mathbf{chunk_1}))\}$	$HA = \{chunk_0, \mathbf{chunk_1}\}$ $HF = \emptyset$
7 `p_global_save=p_global`	$\{(\text{p\_global},(chunk_0)),$ $((\mathbf{EBP_0}, \mathbf{-36}),(\mathbf{chunk_0}))\}$	$HA = \{chunk_0, chunk_1\}$ $HF = \emptyset$
8 `p_global=p`	$\{(\mathbf{p\_global},(\mathbf{chunk_1})),$ $((EBP_0 - 24),(chunk_1))\}$	$HA = \{chunk_0, chunk_1\}$ $HF = \emptyset$
13 `p_global=p_global_save`	$\{(\mathbf{p\_global},(\mathbf{chunk_0})),$ $((EBP_0, -36),(chunk_0))\}$	$HA = \{chunk_0, chunk_1\}$ $HF = \emptyset$
29 `index_user(p_index)`	$\{(\mathbf{p\_global},(\mathbf{chunk_0}, \mathbf{chunk_1})),$ $((EBP_0, -24),(chunk_1)),$ $((EBP_0, -36),(chunk_0))\}$	$HA = \{chunk_0, chunk_1\}$ $HF = \emptyset$
30 `free(p_index)`	$\{((EBP_0, -24),(chunk_1))\}$	$HA = \{chunk_0\}$ $\mathbf{HF} = \{\mathbf{chunk_1}\}$
33 `p_pass=(int*)malloc(..)`	$\{((\mathbf{EBP_0} - \mathbf{28}), (\mathbf{chunk_2}))\}$	$HA = \{chunk_0, \mathbf{chunk_2}\}$ $HF = \{chunk_1\}$
38 `if(*p_pass==*p_global)`	$\{(\text{p\_global},(chunk_0, chunk_1)),$ $((EBP_0 - 28), (chunk_2))\}$	$HA = \{chunk_0, chunk_2\}$ $HF = \{chunk_1\}$

2.2 Value set analysis (*VSA*)

The goal of the value analysis step is to statically discover which program point allocates or frees which heap element. Then, address transfers must be tracked, as well as allocation sizes. Thus, in addition to the two functions HA and HF, our value analysis produces, for each pc, an abstract environment $AbsEnv$. $AbsEnv$ associates to each memory *address* a possible set of *values* this address contains, corresponding either to chunks or allocation sizes.

Table 1 shows some results obtained when analyzing Listing 1. During the two first allocations, lines 22 and 26, two new *chunks* are created. During the call of `index_user` the value of `p_index` is stored, modified and restored (lines 7, 8 and 13). Due to the two branches in `index_user`, possible values for `p_index`, at the end of this call line 29, are $chunk_0$ or $chunk_1$. Thus, at line 30, when $chunk_1$ is freed, `p_index` becomes a dangling pointer. At line 33 a new allocation is placed in `p_pass`, and at line 38 the value of `p_pass` is compared to the value of `p_global`.

Our VSA analysis is a simpler version of existing ones like [4]. Indeed, we focus on information that normally do not require sophisticated numerical computations: finding aliases between memory locations, retrieving heap elements access, and computing allocation sizes. Moreover, it appears that most of the known UaF vulnerabilities are not sensitive to a particular loop iteration. As a result, our analysis is implemented as a forward traversal of the control-flow graph (CFG) of the application, where loops are unrolled at most once, representing several executions of the same loop by a single memory abstraction (location-site abstraction). Moreover, memory modification is implemented using the so-called *weak update hypothesis* [4], in case of approximation.

We give below the transfer functions associated to `malloc` and `free` calls. For a `malloc` call, *ad* denotes the parameter, containing the block size, and *r* denotes the return value. For a `free` call, *ad* denotes the parameter, containing the pointer to be freed. $f \leftarrow \{x \mapsto e\}$ denotes the function identical to f except at point x where the value is e.

Definition 1 Transfer function associated to a malloc call

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

where:

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

and $size(s) = v$ if $s = \{v\}$ and $size(s) = Any$ otherwise[5].

Definition 2 Transfer function associated to a free call

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

where:

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

[5] A better approximation could be provided if it is required for the exploitability analysis.

$(AbsEnv(ad) \cap HE)$ is used because $AbsEnv(ad)$ may refer to some elements that are not in the heap.

3 UaF detection and subgraph extraction

We first explain how the VSA described in the previous section is used to identify UaF patterns, then we show how to extract program slices (as parts of the CFG) to fully characterize each UaF.

3.1 UaF detection

An UaF corresponds to using a dangling pointer. From the results of *VSA* we define $AccessHeap(pc)$, the function that returns all elements of HE that are *accessed* at pc ($AccessHeap : PC \to \mathcal{P}(HE)$). In our implementation, we use REIL[10] as an intermediate representation. In REIL, only two instructions are dedicated to memory accesses:

– `LDM ad,,reg`, to load the content of Mem(`ad`) into the register `reg`
– `STM reg,,ad`, to store the content of the register `reg` into Mem(`ad`)

Therefore, we define $AccessHeap$ as follows:

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

Finally, the set $UafSet$ of all possible UaF vulnerabilities is defined by:

Definition 3 Use after free characterization

$$UafSet = \{(pc, chunk) \mid chunk \in (AccessHeap(pc) \cap HF(pc))\}$$

On the example of Table 1, we have:

$$AccessHeap(38) = \{chunk_0, chunk_1, chunk_2\}$$

and then $UafSet = \{(38, chunk_1)\}$, meaning that $chunk_1$ is dangling and dereferenced at line 38.

3.2 Subgraphs of use after free

The last step of **GUEB** consists in extracting from the initial code a subgraph containing all the instructions involved in a given UaF. Let $pred$ be the function that returns all the predecessors of pc (in a CFG), and $pred^*$ the transitive closure of $pred$. Let also $point\_alloc$ (resp. $point\_free$) be a function that associates to a given chunk the set of pc where this chunk is allocated (resp. freed). Now, for each pair $(pc_{uaf}, chunk_{uaf})$ in $UafSet$, we slice the initial CFG by selecting the following program points (starting from pc_{uaf}):

1. all program points located between the point causing the UaF, pc_{uaf}, and one point that frees $chunk_{uaf}$ (in orange), i.e.,

$$pred^*(pc_{uaf}) \cap succ^*(point\_free(chunk_{uaf}))$$

2. all program points located between one point freeing $chunk_{uaf}$ and the point allocating $chunk_{uaf}$ (in green), i.e.,

$$pred^*(point\_free(chunk_{uaf})) \cap succ^*(point\_alloc(chunk_{uaf}))$$

3. all program points located between the point that allocates $chunk_{uaf}$, and the entry point of the program (blue points), i.e.,

$$pred^*(point\_alloc(chunk_{uaf}))$$

Figure 2 shows the subgraph extracted for our example, at the binary level. This subgraph is useful to study if an UaF can be exploited, and how: for instance if a new allocation takes place between a point where an UaF is freed and dereferenced, as at line 33 in our example.

4 Conclusion

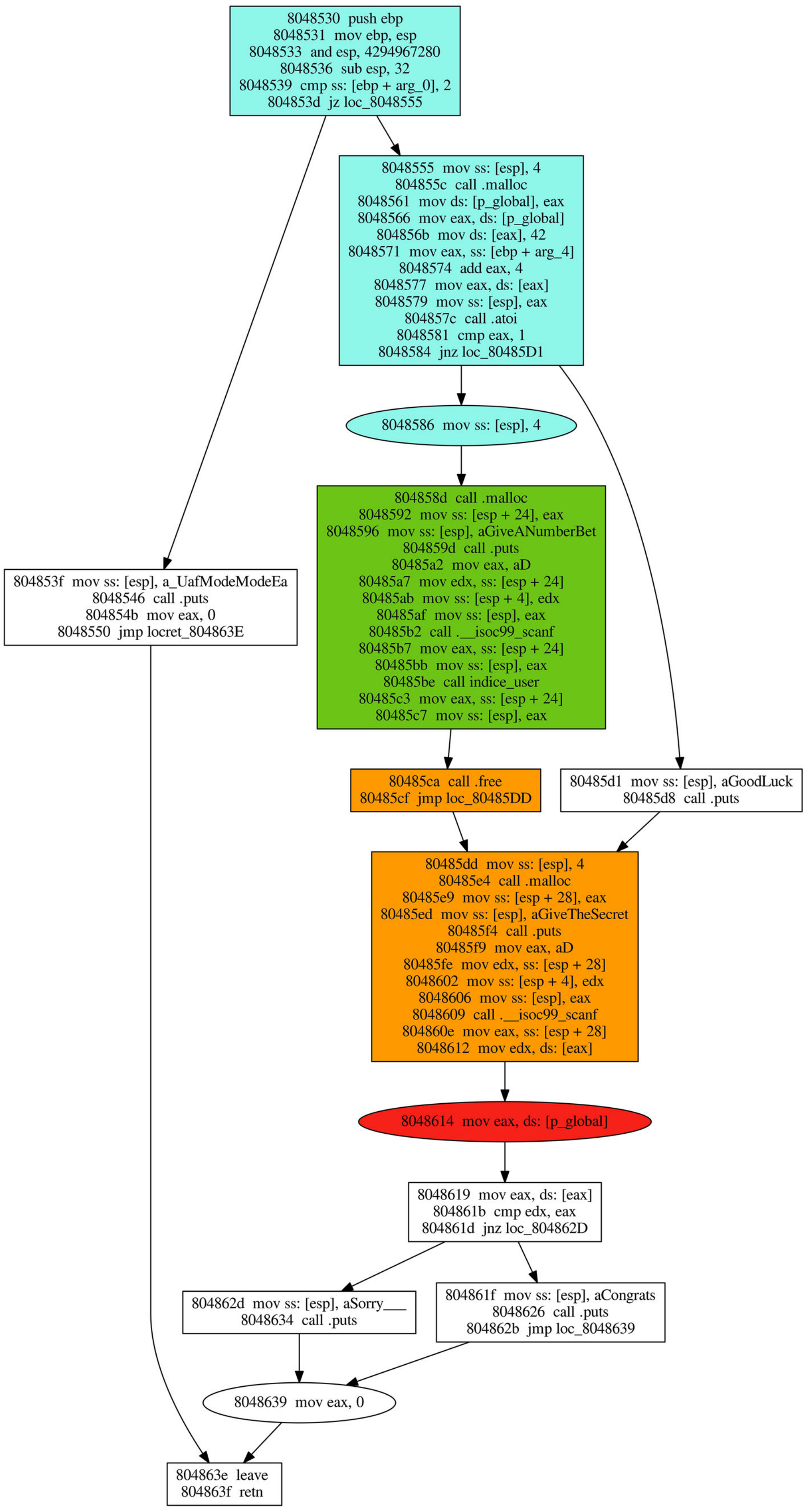
First we give some experimental results obtained, then we discuss some limitations of our current prototype and directions for future work.

4.1 Experimental results

Our approach has been implemented in a prototype tool, **GUEB**, developed in *Jython* and using *IDApro*[6] to transform the binary in assembly code. Then, this assembly code is translated into an intermediate representation, *REIL*[10], using *BinNavi*[7]. Finally we use *Monoreil*, a dedicated API of *BinNavi* allowing to easily implement static analysis on the control flow graph. **GUEB** was evaluated on a real vulnerability, the CVE 2011-4130, appearing in ProFTPD (see [15] for a detailed explanation of this CVE). This form of UaF is close to the one of Listing 1: there exists a path corresponding to an error, where pointers are not correctly restored. From a static analysis point of view, this case study introduces several difficulties: large code size, complex structures, local and global variables, etc. **GUEB** being a prototype, it is relatively slow. To speed up the analysis, we manually selected a subset of 10 functions to be analyzed. Nevertheless this experiment is

[6] https://www.hex-rays.com/products/ida/index.shtml.
[7] http://www.zynamics.com/binnavi.html.

Fig. 2 G_{uaf}



```
8048530  push ebp
8048531  mov ebp, esp
8048533  and esp, 4294967280
8048536  sub esp, 32
8048539  cmp ss: [ebp + arg_0], 2
804853d  jz loc_8048555
```

```
8048555  mov ss: [esp], 4
804855c  call .malloc
8048561  mov ds: [p_global], eax
8048566  mov eax, ds: [p_global]
804856b  mov ds: [eax], 42
8048571  mov eax, ss: [ebp + arg_4]
8048574  add eax, 4
8048577  mov eax, ds: [eax]
8048579  mov ss: [esp], eax
804857c  call .atoi
8048581  cmp eax, 1
8048584  jnz loc_80485D1
```

```
8048586  mov ss: [esp], 4
```

```
804858d  call .malloc
8048592  mov ss: [esp + 24], eax
8048596  mov ss: [esp], aGiveANumberBet
804859d  call .puts
80485a2  mov eax, aD
80485a7  mov edx, ss: [esp + 24]
80485ab  mov ss: [esp + 4], edx
80485af  mov ss: [esp], eax
80485b2  call .__isoc99_scanf
80485b7  mov eax, ss: [esp + 24]
80485bb  mov ss: [esp], eax
80485be  call indice_user
80485c3  mov eax, ss: [esp + 24]
80485c7  mov ss: [esp], eax
```

```
804853f  mov ss: [esp], a_UafModeModeEa
8048546  call .puts
804854b  mov eax, 0
8048550  jmp locret_804863E
```

```
80485ca  call .free
80485cf  jmp loc_80485DD
```

```
80485d1  mov ss: [esp], aGoodLuck
80485d8  call .puts
```

```
80485dd  mov ss: [esp], 4
80485e4  call .malloc
80485e9  mov ss: [esp + 28], eax
80485ed  mov ss: [esp], aGiveTheSecret
80485f4  call .puts
80485f9  mov eax, aD
80485fe  mov edx, ss: [esp + 28]
8048602  mov ss: [esp + 4], edx
8048606  mov ss: [esp], eax
8048609  call .__isoc99_scanf
804860e  mov eax, ss: [esp + 28]
8048612  mov edx, ds: [eax]
```

```
8048614  mov eax, ds: [p_global]
```

```
8048619  mov eax, ds: [eax]
804861b  cmp edx, eax
804861d  jnz loc_804862D
```

```
804862d  mov ss: [esp], aSorry___
8048634  call .puts
```

```
804861f  mov ss: [esp], aCongrats
8048626  call .puts
804862b  jmp loc_8048639
```

```
8048639  mov eax, 0
```

```
804863e  leave
804863f  retn
```

significant enough: we treat a CFG with around 2200 nodes, in 30 minutes on a processor i7-2670QM. We identify the UaF without any false positive and the extracted subgraph is a small slice (it contains 460 nodes).

4.2 Limits and improvements of VSA implementation

In complement to dynamic tools detecting memory errors during executions [12,14], we deliberately choose to use a static approach, in order to address a better coverage of use after free detection. In particular we target small programs (or part of programs) for which high level of security requirements are expected. This static approach can be used to deeply analyze part of programs, detected as sensible thanks to lightweight analysis (including dynamic detection). Nevertheless our VSA is actually not complete and can be improved in several ways. First, the value analysis step can be strengthened: currently loops are expanded at most once, this under-approximation may lead to false negatives (missing some aliases) and gives inaccurate allocation sizes (not really used for the detection step). Taking into account allocation into loops is an open problem both at the source and binary levels [2]. Nevertheless the solution adopted here, consisting in folding all the nodes allocated at a given allocation site (called "allocation-site abstraction" in [2]) appears to be a good trade-off regarding use after free detections. In particular we do not miss UaFs that (generally) only depend on the number of iterations. From a practical point of view, we also need to formalise and develop a more efficient inter-procedural analysis. In this current implementation, we use a *naive* in-lining technique, which does not scale up very well. Using function *summaries* would probably be a better solution. Finally, for compatibility with the REIL framework, **GUEB** is written in *Jython*, which is rather slow. Choosing a more efficient language (such as Caml or C/C++) would significantly speed up our implementation.

4.3 Perspectives

Approaches aiming to precisely analyse exploitability are generally based on symbolic or concolic reasoning, as described in [8,11] in the case of buffer overflow exploitability. The aim we pursue here is to build inputs allowing to exploit use-after-free. Thus, the next step consists in characterizing exploitability of use-after-free, namely the possibility to *modify* and *control* the content of the dangling memory. To do that, we have to identify executions paths in which some new allocations take place between the "free" and the "use" operations, and how this allocated memory can be rewritten from user inputs. This step requires a rather fine-grain heap model, allowing to simulate the allocator behaviour (including potential re-allocations).

References

1. Afek, J., Sharabani, A.: Dangling pointer: pointer. Smashing the pointer for fun and profit. Black Hat USA (2007)
2. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage static analysis. In: Yi, K. (ed.) SAS '06: static analysis symposium, volume 4134 of LNCS, pp. 221–239. Springer, Berlin (2006)
3. Balakrishnan, G., Reps, T.: Wysinwyx: what you see is not what you execute. ACM Trans. Program. Lang. Syst. **32**(6), 23:1–23:84 (2010)
4. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Duesterwald E (edi) CC, volume 2985 of LNCS, pp. 5–23. Springer, Berlin (2004)
5. Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A.: The bincoa framework for binary code analysis. In: Proceedings of CAV'11, pp. 165–170. Springer, Berlin (2011)
6. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: a binary analysis platform. In: Proceedings of the 23rd International Conference on Computer Aided Verification. CAV'11, pp. 463–469. Springer, Heidelberg (2011)
7. Caballero, J., Grieco, G., Marron, M., Nappa, A.: Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Heimdahl, M.P.E., Su, Z. (eds.) ISSTA, pp. 133–143. ACM (2012)
8. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: IEEE Symp. S&P, pp. 380–394 (2012)
9. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c—a software analysis perspective. In: SEFM, pp. 233–247 (2012)
10. Dullien, Thomas, Porst, Sebastian: Reil: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest (2009)
11. Heelan, S.: Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, Computing Laboratory (2009)
12. Nethercote, N., Seward, J.: Valgrind: a program supervision framework. Electr. Notes Theor. Comput. Sci. **89**, 44–66 (2003)
13. Rawat, S., Mounier, L.: Finding buffer overflow inducing loops in binary executables. In: Proceedings of the Sixth International Conference on Software Security and Reliability, SERE 2012, pp. 177–186. IEEE (2012)
14. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: a fast address sanity checker. In: USENIX ATC 2012 (2012)
15. 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