

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/272591534>

Statically detecting use after free on binary code

Article in *Journal of Computer Virology and Hacking Techniques* · August 2014

DOI: 10.1007/s11416-014-0203-1

CITATIONS

43

READS

759

3 authors, including:



Josselin Feist

Verimag

16 PUBLICATIONS 280 CITATIONS

SEE PROFILE

3.4 Laurent Mounier, Marie-Laure Potet, Josselin Feist/ Statically Detecting Use After Free on Binary Code

3.4.1 Marie-Laure Potet

Marie-Laure Potet is professor at Ensimag/Grenoble INP and researcher at Verimag laboratory. Her research interests focus on formal validation techniques, static code analysis and safety/security property modeling and verification.

3.4.2 Laurent Mounier

Laurent Mounier is Maitre de Conferences at Université Joseph Fourier and Verimag laboratory since 1993. His research interests focus on formal validation techniques including model-checking, test and runtime validation, and static analysis.

3.4.3 Josselin Feist

PhD student at Verimag, Grenoble.

3.4.4 Statically Detecting Use After Free on Binary Code

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

- Talk and paper can be downloaded from <http://grehack.org>

Statically Detecting Use After Free on Binary Code^{*}

Josselin Feist and Laurent Mounier and Marie-Laure Potet

Vérimag, centre équation, 2 avenue de Vignate – 38610 Gières, France,
Josselin.Feist@imag.fr, Laurent.Mounier@imag.fr, Marie-Laure.Potet@imag.fr

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 ProFTPD (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. As a result, 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 on 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).

1.1 A motivating example

We explain our approach through a motivating example, given on Listing 1.1¹. The main idea is that the function `index_user` puts `p`, the

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

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

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 area² as pointed to by `p_index`. In this case, the comparison at line 38 will always be true.

Listing 1.1. 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         ;
21         return 0; }
22
23     p_global = (int*) malloc(sizeof(int));
24     *p_global = SECRET_PASS;
25
26     if(atoi(argv[1]) == MODE_EASY) {
27         p_index = (int*) malloc(sizeof(int));
28         printf("Give a number between 0 and 100\n");
29         scanf("%d", p_index);
30         index_user(p_index);
31         free(p_index); }
32     else { printf("Good luck ! \n"); }
33
34     p_pass = (int*) malloc(sizeof(int));
35
36     printf("Give the secret\n");
37     scanf("%d", p_pass);
38
39     if(*p_pass == *p_global) {
40         printf("Congrats ! \n"); }
41     else { printf("Sorry...\n"); }
42     return 0; }
```

² This is the case for instance with the libc

1.2 Our approach

Despite its complexity, static analysis offers several advantages: it supplies more complete results and generally allows to take into account more paths than dynamic analysis. 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 following 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 section 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 section 3.2). Fig. 1 gives the subgraph extracted from Listing 1.1 (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 Polyspace³ 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 mainly based on dynamic approaches. For example, AddressSanitizer[14] uses a specific heap allocator allowing good detection of UaF but does not give information of the root of

³ <http://www.mathworks.fr/products/polyspace>

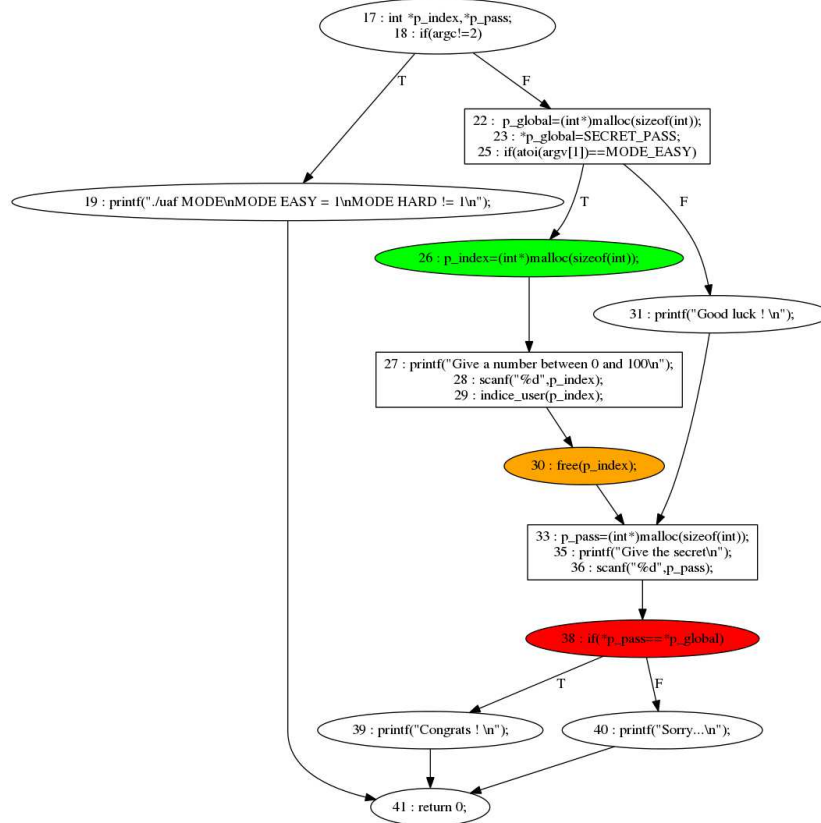


Fig. 1. Extracted Subgraph for example Listing 1.1

vulnerability. 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, section 3 presents the method we propose to detect UaF patterns, and section 4 summarizes some experimental results and gives some limitations of the current prototype and directions for future work.

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 inlining*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, *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* ($HA \in PC \rightarrow \mathcal{P}(HE)$, $HF \in PC \rightarrow \mathcal{P}(HE)$, $\mathcal{P}(S)$ 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).

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 1.1. 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 *chunk₀* or *chunk₁*. Thus, at line 30, when *chunk₁* 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

⁴ making our analysis context-sensitive, but not applicable to recursive calls

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

Table 1. VSA result

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*

$$\begin{aligned} f_{\text{malloc}}(pc, HA, HF, AbsEnv, ad, id_max) &= (HA', HF', r, id_max') \text{ with :} \\ r &= (base_{id_max}, size(AbsEnv(ad))) \\ HF' &= HF \\ HA' &= HA \leftarrow \{pc \mapsto (HA(pc) \cup \{r\})\} \\ id_max' &= id_max + 1 \end{aligned}$$

with $size(s) = v$ if $s = \{v\}$ and $size(s) = \text{Any}$ otherwise⁵.

Definition 2. *Transfer function associated to a free call*

$$\begin{aligned} f_{\text{free}}(pc, HA, HF, AbsEnv, ad) &= (HA', HF') \text{ with:} \\ HF' &= HF \leftarrow \{pc \mapsto (HF(pc) \cup (AbsEnv(ad) \cap HE))\} \\ HA' &= HA \leftarrow \{pc \mapsto (HA(pc) \setminus (AbsEnv(ad) \cap HE))\} \end{aligned}$$

$(AbsEnv(ad) \cap HE)$ 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 ($AccessHeap : PC \rightarrow \mathcal{P}(HE)$). 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:

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

$$\begin{aligned} \text{AccessHeap}(\text{LDM } ad, reg) &= \text{AbsEnv}(ad) \cap HE. \\ \text{AccessHeap}(\text{STM } reg, ad) &= \text{AbsEnv}(ad) \cap HE \end{aligned}$$

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

Definition 3. *Use after free characterization*

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

For example we have $\text{AccessHeap}(38) = \{chunk_0, chunk_1, chunk_2\}$ and then $UafSet = \{(38, chunk_1)\}$, meaning that $chunk_1$ is dangling and dereferenced 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 $point_alloc$ (respectively $point_free$) be a function that associates to a given chunk the set of pc where this chunk is allocated (respectively 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 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 between one point that frees $chunk_{uaf}$, and the point that allocates $chunk_{uaf}$ (in green), i.e:

$$pred^*(point_free(chunk_{uaf})) \cap succ^*(point_alloc(chunk_{uaf}))$$

3. all program points 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.

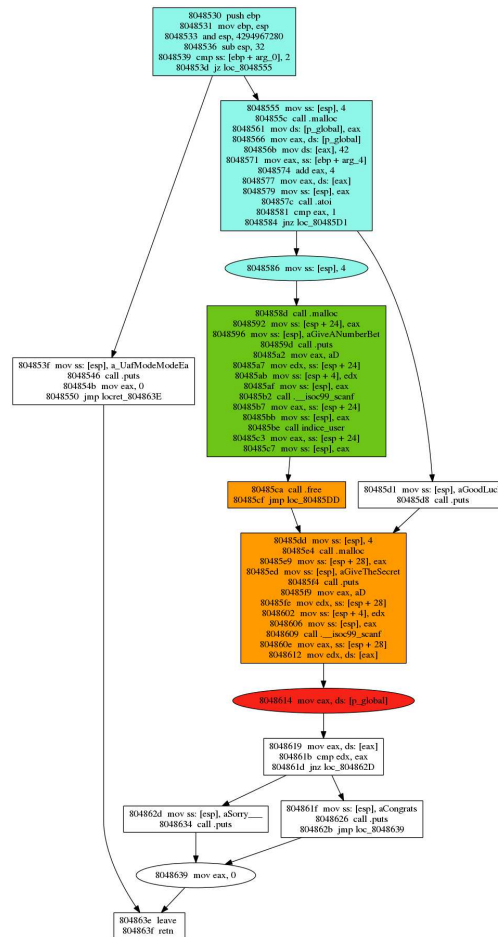


Fig. 2. G_{uaf}

4.1 Experimental results

Our approach has been implemented in a prototype tool, **GUEB**, developed in *Jython* and using *IDApr*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

⁶ <https://www.hex-rays.com/products/ida/index.shtml>

⁷ <http://www.zynamics.com/binnavi.html>

1.1 : 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 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. This static approach can also be used to deeply analyze part of programs, detected as sensible thanks to more 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 interprocedural analysis. In this current implementation, we use a *naive* inlining 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 faster 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 [11,8] 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, that means 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 the allocator behaviour (including potential re-allocations).

References

1. J. Afek and A. Sharabani. Dangling pointer: Pointer. smashing the pointer for fun and profit. Black Hat USA, 2007.
2. Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage static analysis. In Kwangkeun Yi, editor, *SAS '06: Static Analysis Symposium*, volume 4134 of *LNCS*, pages 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.
3. 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.
4. 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.
5. 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.
6. David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification, CAV'11*, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.
7. 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 Zhen-dong Su, editors, *ISSTA*, pages 133–143. ACM, 2012.
8. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symp. S&P*, pages 380–394, 2012.
9. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - a software analysis perspective. In *SEFM*, pages 233–247, 2012.
10. Thomas Dullien and Sebastian Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.
11. Sean Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, 2009.
12. Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89:44–66, 2003.
13. Sanjay Rawat and Laurent Mounier. Finding buffer overflow inducing loops in binary executables. In *Proc of the Sixth International Conference on Software Security and Reliability, SERE 2012*, pages 177–186. IEEE, 2012.
14. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 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.