TÉCNICO LISBOA

HeapDUO: Finding Heap Vulnerabilities in Binary Code

Jorge Cardoso Martins

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Oct 2021

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years. I would also like to thank my family for their understanding and support throughout all these years, to Carolina and Mariana for putting up with me these 4 short years, and to my friends Filipe Casal, Filipe Marques, Paulo Dias, and Vasco Franco. I would also like to acknowledge my dissertation supervisor Prof. Pedro Adão, and Nuno Sabino for their insight, support and sharing of knowledge that has made this thesis possible.

Abstract

Heap memory corruption vulnerabilities are still present in today's software. Since these vulnerabilities are used to gain privileges to numerous widely used systems, it is necessary to find them as quickly as possible. In this thesis we developed HeapDUO, a tool capable of detecting heap vulnerabilities. The name HeapDUO derives from the vulnerabilities it detects, Double-frees, Use-after-frees and Heap Overflows. It also relates to the fact that it is composed by two major components, a static analyser and a symbolic engine.

We extended GUEB [1], a static analysis tool capable of detecting use-after-frees and double-frees in binary code, to also find heap overflows. We improved GUEB's use-after-free detection mechanism, added support for the x86-64 architecture, implemented a more robust memory model, improved the memory usage, and added support for heap buffer overflow detection. To enhance the detection of heap overflows in loops, we combine our static analyser and the symbolic engine AVD [2] to help determine the number of times a loop can be executed. Additionally, and in order to tackle the well known problem of a high number of false positives reported by static analysis tools, we modelled the heap and implemented heap related functions in AVD to be able to triage the vulnerabilities reported by the static analyser.

We evaluated HeapDUO on two different datasets, the Juliet and the CodeQL datasets. In each dataset we compared HeapDUO with GUEB and AFL++ [3], where we analysed the number of vulnerabilities detected by each and also the time it took to analyse them. We also applied HeapDUO to real-world software and detected an already known heap out of bounds read, CVE-2021-32614 [4], in the open-source project dmg2img [5].

Keywords

Static Analysis; Symbolic Execution; Heap Vulnerabilities; Automatic Detection

Resumo

As vulnerabilidades de corrupção de memória da heap ainda estão presentes no software dos dias de hoje. Como estas vulnerabilidades podem ser usadas para obter diversos privilégios, em sistemas amplamente usados, é necessário localizá-las o mais rapidamente possível. Nesta tese desenvolvemos uma ferramenta chamada HeapDUO, capaz de detetar vulnerabilidades de heap, entre elas, double-free, use-after-free e heap overflows. O nome HeapDUO tem como origem as vulnerabilidades que deteta, Double-free, User-after-free e Heap Overflows. O nome da ferramenta está também relacionado com o facto de ser composta por duas componentes, um analisador estático e um motor de execução simbólica.

Neste trabalho estendemos a ferramenta GUEB [1], uma ferramenta de análise estática, capaz de detetar vulnerabilidades de use-after-free e double-free em binários, para também detetar heap overflows. Melhorámos o mecanismo de deteção de use-after-free do GUEB, adicionámos o suporte para a arquitetura x86-64, implementámos um modelo de memória mais robusto e adicionámos suporte para a deteção de heap overflows. Como uma possível fonte de heap overflows ocorre dentro de ciclos, usámos o motor de execução simbólica AVD [2] para determinar o número de vezes que um ciclo pode ser executado. Com esta informação, a análise estática pode assim realizar uma análise mais completa. Além disso, e para lidar com o elevado número de falsos positivos gerados por ferramentas de análise estática, modelámos a heap e implementámos funções da biblioteca relacionadas com a heap no AVD para poder fazer a triagem das vulnerabilidades reportadas pelo analisador estático. A avaliação da nossa ferramenta foi feita recorrendo aos datasets Juliet e CodeQL. Para cada dataset comparámos a nossa ferramenta com a ferramenta GUEB e o AFL++ [3] e para cada uma delas analisámos o número de vulnerabilidades detetadas e também o tempo que demorou para analisá-las. Além disso usámos a nossa ferramenta em programas open-source e fomos capazes de detetar um heap out of bounds read, já conhecido, ao qual foi atribuído o CVE-2021-32614 [4] no projeto dmg2img [5].

Palavras Chave

Análise Estática; Execução Simbólica; Vulnerabilidades de heap; Deteção Automática

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Contents

1.1 Motivation

Software progressively plays a major role in our lives, from web browsers to mobile applications, smart tvs, etc. With the increasing complexity in today's software it is expected that in some particular situations the behaviour of these is different from the expected. When these behaviours decrease the overall security of a system we may have a vulnerability.

Security vulnerabilities can have huge impacts not only for the customers but also to the reputation of companies. For instance Heartbleed [6] was a vulnerability found in OpenSSL cryptographic software library that granted attackers the possibility to steal secret keys used to identify service providers and encrypt traffic, thus efforts to keep software secure are essential. Yet manually securing software by performing code audits is a complex task that takes time and consequently the usage of tools to automatically find vulnerabilities presents itself as a more efficient and probably the only solution for this problem.

Memory corruption bugs are one of the oldest problems in computer security, with their impact ranging from not exploitable to a possible full system compromise. According to MITRE ranking [7], memory corruption bugs are the most dangerous software weakness in 2021. In our work we will focus on these particular class of vulnerabilities.

Usually languages like C and C++ [8] are one of the sources of these type of vulnerabilities because they lack memory security features which allows attackers to alter the program behaviour and even change the control flow. In these languages when the programmer needs to allocate memory at run time it resorts to the heap, and when the requested memory is no longer necessary it is required to manually free it. The fact that the programmer is the one responsible for freeing the memory when he no longer needs it is one of the main sources of heap memory corruption bugs. There are three kinds of heap vulnerabilities: double-free, when the memory is freed twice, use-after-free, when the memory is used after it was freed, and heap overflows, when there are reads and/or writes past the previously allocated memory. To clearly understand how dangerous these are, in Pwn2Own 2021 [9], a computer hacking contest, multiple heap vulnerabilities were used to attack different targets:

1. A heap based buffer overflow was used to get remote code execution on Zoom video client.
2. A double-free was used to perform local privilege escalation on Ubuntu Desktop.
3. A use-after-free was used to perform local privilege escalation on Windows.

1.2 Objectives

The main objective of this thesis is to develop a tool capable of detecting use-after-free, double-free and heap overflow vulnerabilities in binaries without any access to source code information. We will use static analysis on an intermediate language to detect heap vulnerabilities. The use of static analysis may lead to the generation of false positives, meaning that it reports that there are vulnerabilities that in fact do not exist. To filter such reports we will use a symbolic execution engine to verify the presence of the vulnerabilities claimed by the static analyser.

1.3 Contributions

In this work we developed HeapDUO, a tool that detects Double-free, Use-after-free and Overflows on the heap by combining two existing tools, a static analyser, GUEB [1], and a symbolic engine, AVD [2]. We based our work on GUEB, a tool capable of detecting use-after-free and double-free vulnerabilities and made various improvements on it. These improvements include:

- the support for the x86-64 architecture;
- an improvement in the detection of use-after-free vulnerabilities;
- a more robust memory model allowing to reason about buffer overflows;
- the reduction of memory usage; and
- the ability to detect possible heap overflows.

Regarding the symbolic engine, we developed two additional functionalities. The first one with the goal of making our analysis more complete is the ability to inform the static analyser of how many times a loop can be executed based on the information sent by it. The second one is the the ability to detect heap vulnerabilities by developing heap library functions along with heap safety policies which we used to triage the results reported by the static analysis.

We evaluated HeapDUO performance on two different datasets, the Juliet and CodeQL dataset, where we compared the number of vulnerabilities detected by HeapDUO, GUEB and AFL++ and the time it took to analyse them. We also successfully tested HeapDUO's capabilities of detecting previous bugs found by GUEB in real world applications, and were able to detect a recently found heap out of bounds read in an open-source project (CVE-2021-32614 [4]).

1.4 Outline

This thesis is organized as follows: in Chapter 2 we provide a background for the work developed, where we describe symbolic execution and the heap, more specifically the heap internals and the heap memory layout. In this chapter we also present some known heap security vulnerabilities. In Chapter 3 we present GUEB, an open-source static analysis tool that is capable of finding some heap vulnerabilities and in which our work is based on. In Chapter 4 we describe the architecture of our solution and discuss our implementation choices. Chapter 5 presents the evaluation of our work by performing an effectiveness and performance comparison with GUEB and AFL++ using the Juliet and the CodeQL dataset, and also the detection of vulnerabilities in open-source projects. Finally, Chapter 6 concludes our thesis and proposes some directions for future work.

2 Background

Contents

2.1 Symbolic Execution

The main idea behind symbolic execution [10] is to use symbolic variables for inputs instead of actual data, and to represent values of program variables as symbolic expressions. As a result, the output values computed by a program are expressed as a function of the symbolic input values.

A key goal of symbolic execution in software testing is to explore as many different paths as possible. Each path is associated with a set of restrictions on symbolic variables called path conditions. This way it is possible not only to generate a concrete input that reaches these paths, but also check the presence of various kinds of errors such as assertion violations, uncaught exceptions and security vulnerabilities.

Symbolic execution maintains a symbolic state σ, that maps variables to symbolic expressions, and a symbolic path constraint PC which is a quantifier-free first order formula over symbolic expressions.

```
 1  void test(int x, int y) {
 2      int sum = x + y;
 3      if (sum > 32)
 4          ERROR;
 5  }
 6
 7  int main() {
 8      int x = symbolic_input();
 9      int y = symbolic_input();
10      test(x, y);
11      return 0;
12  }
```

Figure 2.1: C program example

For example, symbolic execution of the program in Figure 2.1 starts with an empty the symbolic state σ_0, and the symbolic path constraint PC_0 with $true$. Every time there's a statement $var = symbolic\_input()$ a new unconstrained symbolic value is added to the symbolic state. After line 9 we have $\sigma_0 = \{x \mapsto x_0, y \mapsto y_0\}, PC_0 = \{true\}$.

The symbolic state has to be updated due to the statement in line 2 resulting in $\sigma_0 = \{x \mapsto x_0, y \mapsto y_0\ sum \mapsto x_0 + y_0\}$. In line 3 the conditional expression is added to the path constraint, $PC_0 = \{true \wedge (x_0 + y_0 > 32)\}$, and a new path constraint, PC_1, is created with the symbolic expression negated, $PC_1 = \{true \wedge \neg(x_0 + y_0 > 32)\}$ to account for the else branch of the if. If none of the path constraints can be satisfied then symbolic execution terminates.

When symbolic execution reaches an error it is terminated and it can generate a concrete input by solving the current path constraints, usually the constraints are given to a first order solver. In this example a possible input to reach the error statement could be $\{x = 33, y = 0\}$.

2.1.1 Modern Symbolic Execution Techniques

Classic symbolic execution tries to explore all possible paths of a given program, but following this approach it may be unfeasible to explore all. A simple example is a program that has a loop where the termination condition is symbolic, resulting in infinitely many execution paths. Furthermore there are paths constraints which the first order solver is incapable of solving.

As a result, *Concolic testing* was introduced by mixing concrete and symbolic execution, where symbolic execution is performed dynamically, while the program is executed on concrete input values. Thus whenever the first order solver is unable to solve a particular path constraint concrete values are used releasing the solver from that burden.

Another modern technique used is *Execution-Generated Testing, EGT*, that makes a distinction between concrete and symbolic values by checking before every instruction if all values are concrete. If so the operations are executed as in the original program, avoiding the additional overhead for the symbolic execution.

2.1.2 Symbolic Execution challenges

Symbolic execution faces a few key challenges when processing real-world code, for instance how to deal with path explosion, the environment and constraint solving.

Path Explosion: The most significant challenge regarding scalability is the exponential number of paths in a program due to loops and conditions, since a new instance must be created with a new symbolic state, σ and a new path constraint, PC. To get a general idea a program with N symbolic branches, would generate 2^N instances. As stated in [11], [12] there are some techniques to reduce the state space namely:

- Pruning Unrealizable Paths: At each branch the symbolic execution invokes the first order solver to decide whether the condition given the current path constraint is not satisfiable, so that a new instance doesn't need to be forked.

- Preconditioned Symbolic Execution: The idea is to partition the input space, for instance inputs that don't satisfy some predicate will not be explored. *Known Length* and *Known Prefix* are two possible preconditions [12], where the precondition *Known Length* is specially powerful on loops.

Constraint Solving: Another challenge that stops symbolic execution from scaling is related to the time it takes in constraint solving. Complex programs have complex input constraints, thus it takes more time for the first order solver to provide a solution. To reduce this time, one approach is to apply *Constraint Reduction*, where constraints are reduced into simpler forms and *Reuse of Constraint Solutions*, where queries to the first order solver are cached as well as the solution, so that it can be used in future queries speeding up the process.

Environment: System calls and library functions, influence a program execution, thus these interactions need to be modeled. For instance, KLEE [13] is able to model symbolic files, where these are supported through a basic symbolic file system for each execution state. AEG [12] models symbolic files in a similar approach to KLEE [13], symbolic sockets, environment variables, library function calls and over 70 system calls. DART [14] and CUTE [15] on the other hand, execute external calls by using concrete arguments, therefore some paths will not be explored.

2.2 Heap

A program consists of 4 key memory regions depicted in Figure 2.2: *i) Stack*, grows to lower addresses, works just as a *LIFO* structure and is responsible for holding temporary information such as local variables, return addresses, function arguments, etc., *ii) Text*, includes the instructions that are executed by the program, *iii) Data*, its primary purpose is to hold global and static variables, *iv) Heap*, grows to higher addresses and is responsible for providing dynamically unused memory to the program.

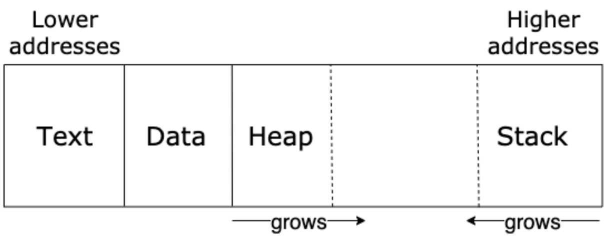


Figure 2.2: User space program memory regions

At runtime programs can request memory dynamically using calls such as *malloc(), calloc(), realloc()* that return a reference to the newly allocated ready to use memory. Since the memory of a computer is limited, when no longer needed, it should be released to be used in further allocations, otherwise the program reaches a state where no memory available is left, preventing it to continue a normal execution. Therefore, something has to be responsible for managing it. This can either be done automatically using garbage collectors or manually, leaving the responsibility on the developer himself, which is the case in languages like C/C++. In Figure 2.3 there is a simple example of a user space program requesting memory dynamically (line 6) and when no longer needed it is freed (line 13).

A variety of heap allocators have been developed to meet specific needs:

- *phkmalloc* used by OpenBSD
- *tcmalloc* developed by Google

```c
int main() {
    int size = 10;
    char *petname = NULL;

    // dynamically request memory
    petname = (char*) malloc(sizeof(char) * size);

    read(stdin, petname, size);
    petname[size - 1] = '\0';
    printf("pet name is %s", petname);

    // free memory when no longer used
    free(petname);
    return 0;
}
```

Figure 2.3: C program dynamically requesting memory

- *jemalloc* used by FreeBSD and Facebook
- *Low Fragmentation Heap* used in Microsoft Windows
- *Hoard* designed to work on several operating systems
- *ptmalloc2* used by GNU LIBC

Heap allocators usually have two main goals: good performance and minimum fragmentation. These are two conflicting goals, since to achieve high performance, operations should be kept to minimum, but to avoid fragmentation it is required additional operations.

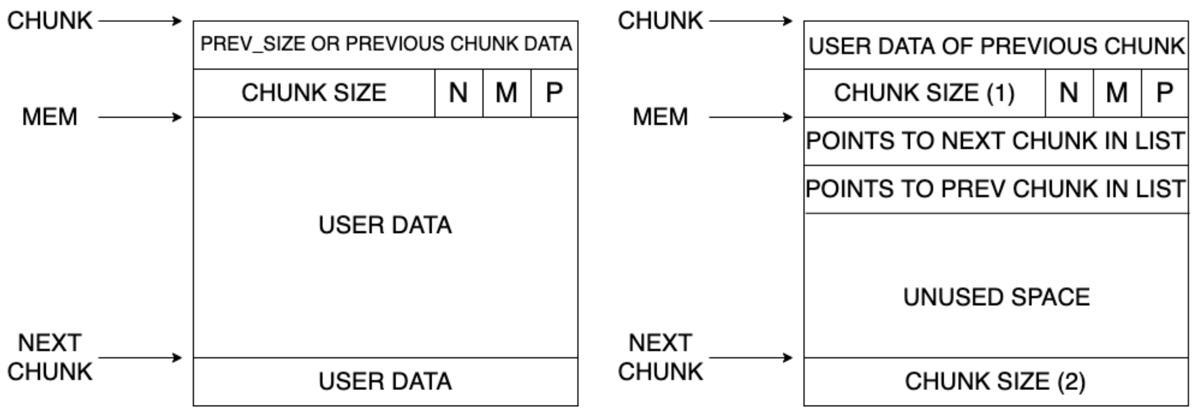
2.2.1 Heap Memory Layout

In this section, we will study *ptmalloc2* (*pthreads malloc*) [16–18], which is used by GNU LIBC [16], GLIBC, and is based on *dlmalloc* [19], due to its importance in linux user space programs.

A heap segment is a contiguous region of memory that is subdivided into *chunks*. A *chunk* is a small region of memory that can be allocated (owned by the program), freed (owned by GLIBC) or combined with adjacent freed chunks into larger ones.

The *ptmalloc2* implementation uses *in-place metadata*, as can be seen in Figure 2.4.

A chunk can be either in two possible states, allocated or freed. Let's start by analysing each state individually for an easier comprehension. Regarding the allocated chunk, Figure 2.4a, the beginning of the chunk contains the header, pointed by the **chunk** arrow. The first field of the header has two meanings based on the previous contiguous chunk state. When the previous contiguous chunk is allocated, then the first field will have data of the previous contiguous chunk, otherwise, the chunk is free thus it will have the size of the previous contiguous chunk.


CHUNK
PREV_SIZE OR PREVIOUS CHUNK DATA
CHUNK SIZE
N
M
P
MEM
USER DATA
NEXT CHUNK
USER DATA
CHUNK
USER DATA OF PREVIOUS CHUNK
CHUNK SIZE (1)
N
M
P
MEM
POINTS TO NEXT CHUNK IN LIST
POINTS TO PREV CHUNK IN LIST
UNUSED SPACE
NEXT CHUNK
CHUNK SIZE (2)

(a) Allocated chunk **(b)** Freed chunk

Figure 2.4: GLIBC chunks

The second and last field of the header, the **chunk size**, contains the size of the chunk in bytes and the last 3 bits give some information about the chunk itself, and if the previous chunk contiguous in memory is allocated or not.

- **prev_inuse (P)** - This bit is set when the previous contiguous chunk is allocated.
- **is_mmaped (M)** - This bit is set when this chunk was mmaped.
- **non_main_arena (N)** - This bit is set when this chunk belongs to a thread arena.

When a program requests memory via *malloc() or calloc() or realloc()*, the reference that is given to the program is the **mem** arrow. This is actually the start of the space reserved for the program to write and read data, which is still part of the chunk. The size of **user data** is always larger or equal to the requested.

As previously mentioned, the first field of a chunk header depends on the previous contiguous chunk state. Since the chunk is allocated, the first field of the next contiguous chunk will contain **user data**, which can be seen by the **next chunk** arrow.

Before analysing the freed chunk, notice that in some cases, depending on the chunks' size, there cannot be two adjacent freed chunks in memory. In those cases, they are merged resulting in a larger chunk. For simplicity, we will assume that all adjacent freed chunks are merged into a single one.

Regarding the freed chunk in Figure 2.4b, since two adjacent contiguous chunks cannot be freed, then the first field of a freed chunk header will always contain **user data** of the previous contiguous chunk, that is allocated. The second field **chunk size** was already explained but now we know that the value of **P** is 1, since it is known that the previous contiguous chunk is allocated.

The *ptmalloc2* keeps track of freed chunks by keeping them in *single linked lists* or in *double linked lists*, according to their size. When a chunk is in a single linked list it uses the first field to point to the next chunk on it and when in a double linked list it uses both fields, one that points to the next chunk and the other to point to the previous. The first field is known as the *forward pointer (fd)* and the second as the *backward pointer (bk)*.

These *single linked lists* and *double linked lists* are called *bins*. Each chunk is inserted in a specific bin according to its size. In *ptmalloc2*, there are 4 types of bins: *i) Fast bin, ii) Unsorted bin, iii) Small bin, iv) Large bin*.

Fast Bin: Although GLIBC supports a maximum of 10 fast bins, the default is to use only 7, where each bin is a *single linked list* of freed chunks. The insertion and deletion happens at the head of the list, working like a *lifo* structure. In each bin, the inserted chunks have all the same size and each fast bin chunks size differs 8/16 bytes in 32/64 bit architectures. For instance the first bin holds chunks of size 16/32, the second holds chunks of size 24/48, and so on, where the last one holds chunks of size 64/128. Note that *fast chunks*, chunks that belong to fast bins, are not combined with adjacent chunks, meaning that they are not merged with adjacent freed chunks. Even though there is a special condition that may force these into merging.

Unsorted bin: When chunks other than *fast chunks*, are freed they are initially stored into a single bin that uses a *double linked list*. The purpose of the *unsorted bin* is to speed up *malloc()* and *free()*, since on deletion it is not necessary to find the appropriate bin, and on allocation if there is a chunk in the *unsorted bin* it can simply be returned to the program, therefore the time to look for the respective bin is eliminated.

Small bin: Chunks with size less than 512/1024 bytes are called *small chunks*. There are a total of 62 small bins where each bin is simply a *double linked list*, where insertion happens at the head of the list and deletion at the end, working like a *fifo* structure. Similiar to *fast bins*, chunks are inserted in a specific *small bin* according to its size. For instance, the first bin holds chunks of size 16/32, the second holds chunks of size 24/48, and so on, where the last one holds chunks of size 504/1008, so they are 8/16 bytes apart. Note that, it is not possible to have two adjacent *small chunks* in memory, when this happens they are both merged and added to the unsorted bin.

Large bin: There are a total of 63 large bins. Chunks whose size is larger than *small chunks* are inserted in these bins. Each bin is a *double linked list* where chunks are stored in decreasing order. Thus insertion and deletion does not happen necessarily at the beginning or end of the list.

Note that all these bins are in the data segment of the GLIBC.

Besides the normal chunks, there are 2 special chunks with a unique purpose, the *top chunk* and the *last remainder chunk*.

The *top chunk* can be considered as a large freed chunk, that does not belong to any bin, and is only used as a last resource, when all available bins are empty. For instance the first program allocation is always served by the *top chunk*. If the size of the *top chunk* is larger than the requested then it will be split in two chunks. The newly allocated chunk, that belongs to the program, and the remainder chunk that becomes the new *top chunk*.

The *last remainder chunk* is the leftover of a small request split. When a program requests memory and its size is qualified as a small chunk, but it cannot be served by the *small bin* nor the *unsorted bin*, the *large bin* is used. If there is an available large chunk it is split in two, the chunk that belongs to the program and the remainder chunk that gets added to the unsorted bin. This remainder chunk becomes the *last remainder chunk*. Therefore on upcoming small chunk requests, if the *last remainder chunk* is the only chunk in the *unsorted bin* it will be split in two once again, helping achieving locality of reference due to the memory allocations being next to each other.

2.2.2 Thread Local Cache

Thread Local Cache, tcache, is a new caching mechanism introduced in GLIBC 2.26, 2017. The main objective is to improve the heap management performance by allowing threads to allocate memory at the same time, thus not needing to acquire locks since each thread now has its own free list data structures. Notice that when a thread free list structure is empty, it still resorts to the structures introduce before, such as *fast bin, unsorted bin, small bin* and *large bin*. Tcache introduces two new structures, Figure 2.5.

```c
#define TCACHE_MAX_BINS 64

typedef struct tcache_entry
{
  struct tcache_entry *next;
} tcache_entry;

typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

Figure 2.5: tcache structures

Tcache is essentially an array of *single linked lists* denoted by *entries*, where each *tcache_entry* contains linked chunks of a certain size. Also each entry has a maximum of 7 linked chunks, which is kept by the array *counts*, that keeps track of how many chunks are linked.

Insertion and deletion happens at the head of the list just like in *fast bins*, but on the contrary the *next* pointer, points to the **user data** of the chunk and not to the beginning.

In Figure 2.6, there is a simple visual representation of the tcache. The first tcache bin, at index 0, has a single chunk, the second bin has no elements and the third bin, at index 2, has 2 chunks.

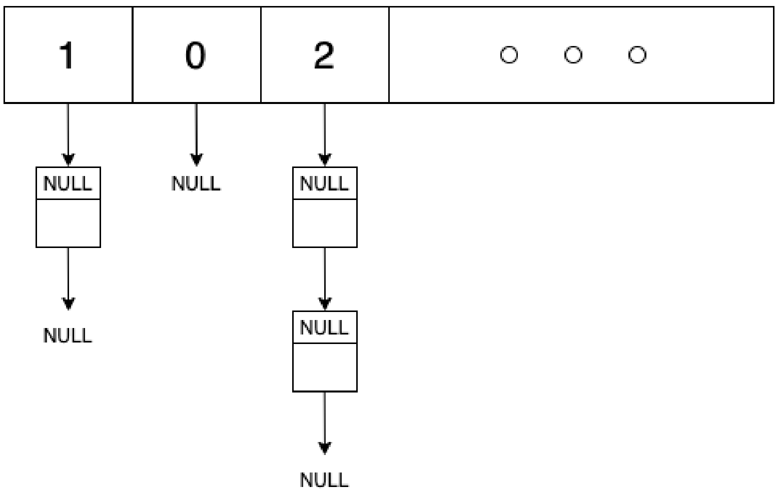


Figure 2.6: tcache visual representation

2.2.3 Heap vulnerabilities

In this section we will present the heap vulnerabilities that will be considered in this work.

```
1  int main {
2      char *buf = (char*) malloc(0x10);
3      // ...
4      if (condition)
5          free(buf);
6          // return 0;
7
8      struct function_pointers* fp = (char*) malloc(0x10);
9      // function_pointers initialization
10
11     // Use-after-free, if condition = true
12     read(0, buf, 0x10);
13     fp->foo();
14     return 0;
15 }
```

Figure 2.7: Use-after-free

Use-after-free, known as UAF, presented in Figure 2.7 is a heap vulnerability that occurs when a program uses memory that had been previously freed. In this example, the program requests memory from the heap and performs some computations with it. If *condition* is evaluated to true, then the chunk allocated in line 2, will be placed in the tcache bin that holds chunks of size `0x10`. In line 7 the program requests a memory region of the same size, `0x10`, which *malloc()* will end up returning to the program the same memory that was freed in line 5. Notice that this memory is now initialized with function pointers. The vulnerability occurs at line 12, where data is being read to the memory pointed by *buf*. Since *buf* is a reference to the memory which is now initialized with function pointers, then the user is able to control the entire function pointers structure, thus being able to control *fp→foo* and redirect code execution at line 13.

This is a very simple example, just to get an insight of the consequences of such a vulnerability. In reality, even without the *function pointers* structure it is still possible to corrupt the program data, make a program crash acting like a denial of service attack, possibly construct arbitrary read/write primitives or even get arbitrary code execution.

Double-free is a special case of Use-after-free, which occurs when a pointer is freed more than one time. This vulnerability may lead to undefined behaviour but it can also be exploited to construct arbitrary memory read/write primitives and possibly leading to arbitrary code execution.

```c
int main {
    char *buf = (char*) malloc(0x10);
    free(buf);
    free(buf); // double-free
    return 0;
}
```

Figure 2.8: Double-free

```c
#define SIZE 32
int main() {
    char* src = NULL;

    // initialize src pointer ...

    char* buffer = (char*) malloc(sizeof(char) * SIZE);

    // heap overflow: off by one
    for (int i = 0; i <= SIZE; i++)
        buffer[i] = src[i];

    return 0;
}
```

Figure 2.9: Heap overflow

Heap Overflow is similar to a buffer overflow. It happens when the size of the data written is larger than the size of the chunk. If the data is user controlled then the user can change adjacent chunk metadata, such as the chunk size, the *prev_inuse* bit and the actual data stored by it. In case the adjacent chunk is freed, the user can change the pointers from the linked list to its own advantage.

2.3 Related Work

2.3.1 Symbolic execution

There are already a variety of tools capable of performing concolic symbolic execution.

KLEE [13] is a symbolic execution engine with 2 main goals: *i)* hit every line of executable code, *ii)* at each operation detect if any input value can cause an error, like dereferences, assertions, buffer overflow, etc. When *KLEE* finds a possible error it is able to solve the current path constraints and create a test case for it, thus it is able to generate tests that achieve high code coverage on a set of complex programs. *KLEE* was also used as a bug finding tool being able to detect 56 bugs over a total of 452 different applications.

DART [14] was first implemented at Bell Labs for testing C programs. It combines dynamic test generations with random testing and model checking techniques with the goal of executing all the possible paths of a programs, while still checking for possible errors as program crashes, assertion violations and non-termination. One of the programs tested by *DART* was *oSIP* [20], an open source implementation of *SIP*, Session Initiation Protocol, where it was able to crash about 65% of the *oSIP* functions, most of them due to null pointer dereferences.

CREST is an open-source tool for concolic testing of C programs, that can use different search path heuristics, in order to scale for larger programs. In [21], the authors, propose new search heuristics such as control flow directed search, uniform random search and random branch search. With these 3 new different techniques *CREST* was able to achieve more branch coverage when comparing to a bounded DFS search.

CUTE [15] ”is a concolic unit testing engine that extends *DART* to handle multi-threaded programs that manipulate dynamic data structures using pointer operations”, [10]. When applied to *SGLIB* [22], a C data structure library, it found 2 bugs, a segmentation fault and an infinite loop.

2.3.2 Symbolic Execution Applied to Security

Early work on automatic exploit generation focused primarily on stack-based overflows.

In AEG [12], the authors developed a tool capable of finding vulnerabilities on source code and generating exploits for them, showing that exploit generation for control flow hijack attacks can be modelled as a formal verification problem. The authors introduced a new technique, *preconditioned symbolic execution* that narrows down the state space to be explored and implemented 4 different preconditions:

- *None*: The state space is explored normally.
- *Known Length*: The size of the symbolic input is known, thus reducing the state explosion when executing a loop.

- *Known Prefix*: The prefix of the symbolic input is known. This precondition helps when targeting programs with a specific protocol or format, for instance parsing images.
- *Concolic Execution*: In case there is an input that crashes the program, it can be used to determine if it can be exploitable.

Regarding the vulnerabilities targeted, only stack buffer overflows and format strings are considered allowing to perform *return-to-stack* and *return-to-libc* attacks, thus not taking into consideration possible binary protection schemes, such as NX and ASLR.

The basic exploitation principles introduced in AEG [12], were applied to MAYHEM [23] the winner of DARPA Cyber Grand Challenge [24]. The vulnerabilities considered by MAYHEM were the same as in AEG.

HeapHopper [25] is a system that uses dynamic symbolic execution to find weaknesses in heap allocators implementation. HeapHopper needs a set of heap operations, *malloc*, *free*, *Overflow*, *Double-free*, *Use-after-free*, to create a list of interactions, that are used to identify security violations, which are detected while executing symbolically each heap allocator implementation. The security violations considered by HeapHopper are:

- *Overlapping Allocations* - when *malloc()* returns memory that is already allocated.
- *Non-Heap allocation* - when *malloc()* returns memory that is not inside the heap boundaries
- *Arbitrary write* - whether is its possible to write any content to an arbitrary location

ArchHeap [26] essentially tries to achieve the same as HeapHopper, but it is independent of the heap implementation. To be heap implementation independent the authors performed an analysis of different heap allocators and found common designs, such as *binning* and *in-place metadata*. With this common designs an heap model abstraction was created, avoiding the need of symbolic execution. ArchHeap showed by its results that was able to outperform HeapHopper since it was able to find new exploitation primitives.

2.3.3 Fuzzing

Fuzzing is a widely used technique in the security field which has demonstrated to be an effective way of finding software vulnerabilities automatically. The general idea behind it is to feed the target program with multiple randomly mutated inputs and monitor the target for a possible crash. Now a days most fuzzers such as AFL++ [3], an improved version of AFL [27], take advantage of a feedback mechanism to decide whether a given mutated input is interesting and should be kept for additional mutations. An input is considered to be interesting when it reaches previously unexplored points in the target program, which increases the program coverage. AFL++ has proved its capabilities by detecting multiple vulnerabilities in open-source projects.

3 GUEB Graphs of Use-After-Free Extracted from Binary

Contents

In this section we describe Graphs of Use-After-Free Extracted from Binary (GUEB) from Josselin Feist [1], with the goal of triggering Use-After-Free using automated program analysis techniques on binary code. Our work will be based on GUEB which we extend to support better use-after-free detection and heap overflow detection.

3.1 Value Set Analysis

Value Set Analysis (VSA) is a program analysis algorithm that determines an over-approximation of the set of numeric values and addresses that each register and memory location holds at each program point [28].

In the context of GUEB, the analysis of a program always starts at a function entry-point, usually *main*, where all instructions begin with an empty memory state. When analysing an instruction its memory state is changed, and this new memory state is propagated to the memory states of all the next instructions to be analysed. The process is repeated until all the instructions are analysed.

In every program the existence of control flow instructions determines which instructions will be executed next based on a given condition, which requires a merge operation between two memory states, since certain instructions can be reached from different paths. Also the same instruction can be executed several times due to loops. In these cases they are analysed until the memory state converges.

In Figure 3.1, we have the pseudo-code of a program to demonstrate how the merge operation is used when certain instructions can be reached from several paths. In this case the instruction at line 4, can be reached directly from the instruction at line 1 or at line 3. Recalling the memory state, we can say with certainty that at, *a = 0*, the memory state contains the variable *a* equal to zero and at *a = 1* the variable is updated to one. Now regarding line 4, it is only possible to infer the content of the variable *a* based on the value of the *condition*. Since the author considered the analysis to be path insensitive, meaning all paths are evaluated regardless of the path conditions, the memory state at line 4 has both values, $a = \{0, 1\}$.

```
1 a = 0
2 if (condition)
3     a = 1
4 ...
```

Figure 3.1: Value Set Analysis

The process of how memory states are propagated is discussed in greater detail in Section 3.4.

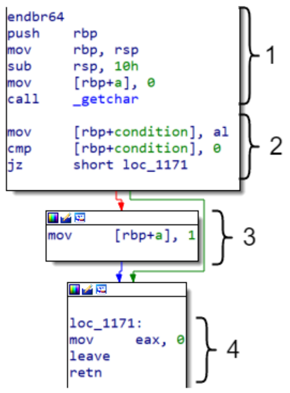
3.2 Static analysis at machine code level

In [1], the author highlights several **key concepts** required for GUEB to statically analyse a binary.

I) Control Flow Graph recovery - The first step when analysing a binary program, is to disassemble the binary code and to build a control flow graph, which stores information about all possible execution paths that the program can take. GUEB relies on a disassembler called IDA [29], that given a binary program it generates the assembly code and the control flow graph for every function. In Figure 3.3, we can see the resulting control flow graph of the program in Figure 3.2, which also gives information regarding the relation between different basic blocks (sequence of instructions with no branches), which GUEB heavily relies on to detect double-free and use-after-free vulnerabilities. In this case there are 4 basic blocks, since only the last instruction of a basic block can cause code to be executed in a different one.

```c
int main() {
    int a;
    char condition;

    a = 0;
    condition = (char) getchar();

    if (condition)
        a = 1;

    return 0;
}
```

Figure 3.2: Source code



```
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+a], 0
call    _getchar

mov     [rbp+condition], al
cmp     [rbp+condition], 0
jz      short loc_1171
```

```
mov     [rbp+a], 1
```

```
loc_1171:
mov     eax, 0
leave
retn
```

Figure 3.3: Control Flow Graph

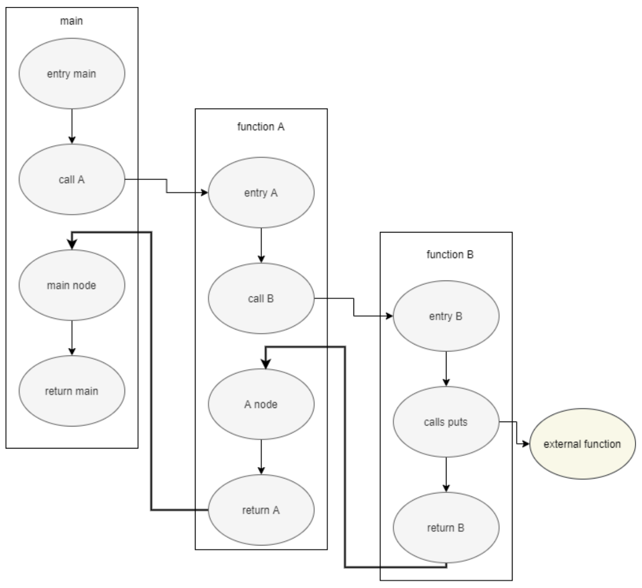
II) Loop detection and unrolling - As stated before, loop instructions can be executed several times. To perform VSA on these instruction we can execute them until fixed-point is reached, however in [1] the author decided that loops could be unrolled a fixed number of times. As the author states *"Although not correct, this technique showed good results in practice"* [1]. To unroll a loop, the loop structure needs to be recognized using it's control flow graph, by identifying the existing strongly connected components (SCC), and then simply unrolling these SCC.

The algorithm to find and unroll loop structures an arbitrary number of times is presented in [1], so the important thing to keep in mind is that we have access to an algorithm that is capable of unrolling loops, which will be later explored in Section 4.3.2.

III) Function inlining - In real world scenarios the location in the code of the events that trigger a vulnerability, such as calling `malloc()`, `free()` and triggering the vulnerability itself are not often located in a single function, on the contrary, these events usually happen several functions apart. So in order to have good results, *interprocedural* analysis was implemented where the values of global variables or

```
void B() {
    puts("hello");
}
void A() {
    B();
}
int main() {
    A();
    return 0;
}
```

(a) Source Code



(b) Inlined CFG's

Figure 3.4: Function inlining

variables passed to a function are considered when analysing it, enabling a more precise analysis, as opposed to *intraprocedural* analysis, where the scope of each variable in a function is limited to only that function. The technique of function inlining was used to perform *interprocedural* analysis.

For instance in Figure 3.4a, *main* calls function *A* which then calls function *B*. When analysing *main*, and when the instruction that calls function *A* is reached, the memory state of the callee instruction is passed as an initial memory state for *entry A* first instruction. When function *B* is called the same procedure happens, eventually reaching the *return B* node, that has a new memory state that contains function *B* changes. When returning from function *B*, the final memory state of *B* is propagated back to the instruction that follows the one that called *B*, and the same process happens when reaching the *return A* node. Notice that function *B* calls *puts*, which does not have an implementation available in the program since it is a GLIBC function. In these cases, where there are calls to external functions that do not affect our analysis, their effects are simply ignored by not considering the function call. We only take into consideration calls to external functions that are required for our analysis such as `malloc()` and `free()`.

3.3 Memory Model

VSA is a program analysis algorithm that determines an over-approximation of the set of numeric values and addresses that each register and memory location holds at each program point [28].

As stated before, in a program memory space Figure 2.2, there are 3 main sources of data, the stack, the heap and the data region. Intuitively, VSA would split the memory space of a program into the same memory regions, yet the author defines 5 disjoint memory regions:

I) Global region: Memory locations representing the *data* address space of a program.

II) Heap region: Memory locations representing the *heap* address space of a program.

III) Register region: Memory locations representing registers.

IV, V) $\textbf{Init}_{\textbf{reg}}$, $\textbf{Init}_{\textbf{mem}}$: Memory locations representing values and addresses that were not initialized.

Furthermore, associated with VSA, there is the concept of value set, memory location, and abstract environment:

- **value set** — the set of all possible addresses and values
- **memory location**, *memLoc* — represents a memory address that contains a **value set**
- **abstract environment**, *absEnv* — function that maps any **memory location** to all its possible values, the **value set**

A memory location is defined by the following grammar in Figure 3.5:

$$
\begin{array}{rcl}
\langle\text{addr}\rangle & ::= & \mathbb{N} \\
\langle\text{offset}\rangle & ::= & \mathbb{Z} \\
\langle\text{chunk}\rangle & ::= & \textit{string} \\
\langle\text{init\_reg\_name}\rangle & ::= & \textit{string} \\
\langle\text{init\_mem\_name}\rangle & ::= & \textit{string} \\
\langle\text{reg\_name}\rangle & ::= & \textit{string} \\
\langle\text{heap}\rangle & ::= & \textbf{\textit{He}}\ \langle\text{chunk}\rangle \\
\langle\text{memLoc}\rangle & ::= & \textbf{\textit{Globals}}\ \langle\text{addr}\rangle \ \mid\ \textbf{\textit{Registers}}\ \langle\text{reg\_name}\rangle \ \mid\ \langle\text{heap}\rangle\times\langle\text{offset}\rangle \\
 & & \mid\ \textbf{\textit{Init}}_{\textbf{\textit{reg}}}\ \langle\text{init\_reg\_name}\rangle\times\langle\text{offset}\rangle \ \mid\ \textbf{\textit{Init}}_{\textbf{\textit{mem}}}\ \langle\text{init\_mem\_name}\rangle\times\langle\text{offset}\rangle \\
 & & \mid\ \top
\end{array}
$$

Figure 3.5: Memory location grammar

For instance we represent a heap memory location with $\{$**He**(”chunk1”), 0x0$\}$ where ”chunk1” is a string and 0x0 is the offset. Note that $\top$ represents all possible memory locations.

The value set is defined by the following grammar in Figure 3.6:

$$
\begin{aligned}
\langle\text{offsets}\rangle &::= \langle\text{offset}\rangle \mid \langle\text{offset}\rangle + \langle\text{offsets}\rangle \\
\langle\text{base}\rangle &::= \textbf{\textit{Constant}} \mid \langle\text{heap}\rangle \mid \textbf{\textit{Init}}_{\textbf{\textit{reg}}}\ \langle\text{init\_reg\_name}\rangle \mid \textbf{\textit{Init}}_{\textbf{\textit{mem}}}\ \langle\text{init\_mem\_name}\rangle \\
\langle\text{value}\rangle &::= \langle\text{base}\rangle \times \langle\text{offsets}\rangle \\
\langle\text{values}\rangle &::= \langle\text{value}\rangle \mid \langle\text{value}\rangle + \langle\text{values}\rangle \\
\langle\text{valueSet}\rangle &::= \langle\text{values}\rangle \mid \top
\end{aligned}
$$

Figure 3.6: Value set grammar

In the value set definition, the symbol + represents concatenation, for instance, *offsets* is defined as a single number or as multiple numbers, *offsets* $= [-1]$ or *offsets* = $[-1, 4, 5]$.

Often in pointer based languages we can dereference a pointer and treat its value as another memory location, just like in Figure 3.7 at line 17. With this case in mind this memory model needs to support these cases, meaning converting a value set into a memory location.

The author defines 2 functions for dereferencing pointers:

1. `value_to_loc_set` that takes as input a *value* = *base* $\times$ *offsets* and outputs a set of memory locations.

2. `value_set_to_loc_set` that takes as input a *valueSet* = *values* $\mid$ $\top$ and outputs a set of memory locations

$$
\texttt{value\_to\_loc\_set}(\text{base}, X) = \begin{cases} \{\textbf{Globals}(x) \mid x \in X\}, & \text{if base = } \textbf{Constant} \\ \{\textbf{Init}_{\textbf{reg}}(n, x) \mid x \in X\}, & \text{if base = } \textbf{Init}_{\textbf{reg}}\text{(n)} \\ \{\textbf{Init}_{\textbf{mem}}(n, x) \mid x \in X\}, & \text{if base = } \textbf{Init}_{\textbf{mem}}\text{(n)} \\ \{\textbf{He}(n, x) \mid x \in X\}, & \text{if base = } \textbf{He}\text{(n)} \end{cases}
$$

$$
\texttt{value\_set\_to\_loc\_set}(V) = \begin{cases} \bigcup\limits_{v=V} \texttt{value\_to\_loc\_set}(v), & \text{if } V \neq \top \\ \{\top\}, & \text{otherwise} \end{cases}
$$

Now that we have defined the memory model, we can finally understand how this model works, by analysing a single instruction. The main goal of this example is to explain the use of $\textbf{Init}_{\textbf{reg}}$ and memory dereferences. In most functions the two first instructions are `push rbp` and `mov rbp, rsp` followed by `sub rsp, offset` in case it is necessary to reserve space for local variables.

The same happens for the first instructions of the program Figure 3.7, which are `push rbp; mov rbp, rsp; sub rsp, 0x10`, where it is subtracted the value `0x10 = 8 * 2` from the stack pointer to allocate space on the stack for the two pointers `char* name` and `struct person* person`. From the first two instructions we can see that it is assumed that `rbp` and `rsp` already have concrete values, thus the need for $\textbf{Init}_{\textbf{reg}}$. At the beginning of the analysis of a program it is required to initialize the values of both `rsp` and `rbp`, so even before analysing the first instruction of main we already initialized our memory state with:

Registers(rsp) $\rightarrow$ {$\textbf{Init}_{\textbf{reg}}$(rsp), {0x0}} ; **Registers**(rbp) $\rightarrow$ {$\textbf{Init}_{\textbf{reg}}$(rbp), {0x0}}

Now, we are ready to start analysing the first instruction `push rbp`, which decrements the current value of `rsp` and writes into the memory it points to the current value of `rbp`. Note that this can be seen as a memory dereference since we treat the contents of `rsp` as a memory location. So we first decrement the value set of **Register**(rsp) by 8 and we get:

Registers(rsp) $\rightarrow$ {$\textbf{Init}_{\textbf{reg}}$(rsp), {-0x8}}

Then we convert the value set of **Register**(rsp) to a memory location using the function previously defined `value_set_to_loc_set`, with $\textbf{Init}_{\textbf{reg}}$(rsp), {- 0x8} as input, returning the memory location $\textbf{Init}_{\textbf{reg}}$(rsp, -0x8). The last step is to write at the new memory location the contents of the **Register**(rbp):

$\textbf{Init}_{\textbf{reg}}$(rsp, -0x8) $\rightarrow$ {$\textbf{Init}_{\textbf{reg}}$(rbp), {0x0}}

Taking everything into account we get the following memory state from just executing the first instruction in the program:

Registers(rsp) $\rightarrow$ {$\textbf{Init}_{\textbf{reg}}$(rsp), {-0x8}} ; **Registers**(rbp) $\rightarrow$ {$\textbf{Init}_{\textbf{reg}}$(rbp), {0x0}}

$\textbf{Init}_{\textbf{reg}}$(rsp, -0x8) $\rightarrow$ {$\textbf{Init}_{\textbf{reg}}$(rbp), {0x0}}

Fast forwarding into analysing the instruction at line 16 of Figure 3.7, that generates the assembly instructions `mov rax, [rbp - 0x8]; mov rdx, [rbp - 0x10]; mov [rax], rdx` and considering the following variable mapping:

- `name` = $\textbf{Init}_{\textbf{reg}}$(rbp, -0x10) $\rightarrow$ {**He**("chunk1"), {0x0}}
- `person` = $\textbf{Init}_{\textbf{reg}}$(rbp, -0x8) $\rightarrow$ {**He**("chunk2"), {0x0}}

After the first two instructions, we get as a result:

Register(rax) $\rightarrow$ {**He**("chunk2"), {0x0}} **Register**(rdx) $\rightarrow$ {**He**("chunk1"), {0x0}}

The last instruction is the one that initializes the `struct person` with a pointer to its name, this is done by converting the value set of `rax` into a memory location, again using `value_set_to_loc_set`, and writing to it the value set of `rdx`. After this instruction this is the first time that the memory location **He** has a value set:

He("chunk2", 0x0) $\rightarrow$ {**He**("chunk1", {0x0})}

In the `// initialize name` at line 14 of Figure 3.7, if we had the following instruction `name[5] = 'A';` we would get additionally:

He("chunk1", 0x5) $\rightarrow$ {**Constant**, {0x41}}

```c
struct person {
    char* name;
    // ...
};

int main() {

    char* name = NULL;
    struct person* person = NULL;

    name = (char*) malloc(sizeof(char) * 32);
    person = (struct person*) malloc(sizeof(struct person));

    // initialize name ...

    person->name = name;
    puts(person->name);

    return 0;
 }
```

Figure 3.7: Pointer dereference

3.4 GUEB VSA algorithm

As stated before, a control flow graph is composed by multiple basic blocks connected to others establishing a parent/child relation, where each basic block has multiple instructions without any branches. For every instruction in a control flow graph there is a memory state. This state, is propagated to the following instructions therefore each instruction has a State_{in} and a $\text{State}_{\text{out}}$. The State_{in} acts like the memory state before a instruction is analysed and the $\text{State}_{\text{out}}$ contains the changes resulting from analysing the given instruction. In general the State_{in} of one instruction is exactly the $\text{State}_{\text{out}}$ of the previous instruction – for instance in Figure 3.3 the State_{in}(`cmp [rbp + condition], 0`) is the same as the $\text{State}_{\text{out}}$(`mov [rbp + condition], al`). The State_{in} can also be a composition of multiple $\text{State}_{\text{out}}$ – for instance the instruction `mov eax, 0` also in Figure 3.3 has to take into account both $\text{State}_{\text{out}}$(`mov [rbp + a], 1`) and $\text{State}_{\text{out}}$(`jz short loc_1159`), since the analysis is path insensitive.

Algorithm 3.1: analyse_func(F, State_{in})

```
    $entry\_bb \longleftarrow get\_entry\_bb(F)$
    $analyse\_func\_bbs(entry\_bb)$
    **return** $merge\_states(\{State_{out}(n) \mid n \in return\_insts(F)\}$
```

In a high level way the analysis starts by initializing the frame registers, (`rsp, rbp`), and calling `analyse_func` Algorithm 3.1 on the entry function of the program, with the initialized state as an argument. Since it is possible that a given basic block can have multiple parents and we only want to analyze it once, we make use of the function `already_analysed` that tells if a given basic block was already analysed by checking if the last instruction's $\text{State}_{\text{out}}$ is not empty. Also this basic block can only be analysed when all the parent basic blocks have already been analysed, which is checked by the function `all_parents_analysed`. Regarding the analysis of the instructions, we say an instruction can be one of two types: a `call` instruction in which case we call `handle_call` Algorithm 3.4 or it can be a regular instruction that is analysed by `function_transfer`. Note that `ignore_call`, resets the stack pointer, the `rsp` register, to its previous value, since a `call` instruction pushes the next instruction address to be executed after the function to the stack.

The analysis is performed on top of an intermediate language, Reverse Engineering Intermediate Language, *REIL* [30], that can translate instructions from different architectures such as x86, PowerPC-32 and ARM-32. It has a simple instruction set with only 17 different instructions and it has infinite temporary registers, *t0, t1,* etc.

The syntax of *REIL* instructions is of the following form:

$$\texttt{opcode } (\texttt{R}_\texttt{A}, \ \texttt{R}_\texttt{A} \texttt{ size}), \ (\texttt{R}_\texttt{B}, \ \texttt{R}_\texttt{B} \texttt{ size}), \ (\texttt{R}_\texttt{C}, \ \texttt{R}_\texttt{C} \texttt{ size})$$

The source are registers `A` and `B` and the destination is register `C`. The size operand can be represented by `b1/b2/b4/b8/b16` which denotes the size of the operand in bytes, For example we can have:

- `add (t1, b4), (4, b4), (rax, b8)` - adds 4 to the contents of the temporary register `t1` and stores the result in `rax`.

- `stm (t1, b4), , (t2, b4)` - stores the contents of the temporary register `t1` in the memory location given by `t2`

Note that GUEB ignores the operand size however in our solution we use it as described in Section 4.2.3.

The `transfer_function` indicates the effects of analysing each *REIL* instruction on the memory state and it is the only function that is dependent on the intermediate language. The only instruction that needs special attention is the *store to memory*, `stm`. Considering the example, `stm (t1, b4), , (t2, b4)`, the first step is to convert the value set of `t2` into a memory location, using the already described function `value_set_to_loc_set`. At this point we need to consider four different possibilities regarding the resulting memory location:

1. There is only one valid memory location

2. There are multiple possible memory locations

3. The memory location is $\top$ (top)

4. The number of memory locations exceeds a value predetermined

The first case is called a *strong memory update*, since there is only one possible memory location, so the contents of the register `t1` are stored in it. In the second case we have more than one possible memory location, which is called a *weak memory update*. Figure 3.8 illustrates an example where such a situation can happen. In line 17 `current_buffer` can hold two different values, `buffers[0]` or `buffers[1]`, thus we call this memory store a *weak memory update*.

```
 1  char* buffers[2];
 2
 3  int main() {
 4
 5          // init buffers array
 6          buffers[0] = (char*) malloc(sizeof(char) * 10);
 7          buffers[1] = (char*) malloc(sizeof(char) * 10);
 8
 9
10          char* current_buffer = buffers[0];
11
12          char option = getc(stdin);
13          if (option == 'N') {
14                  puts("buffer1 selected");
15                  current_buffer = buffers[1];
16          }
17          current_buffer[0] = 'A';
18
19          return 0;
20  }
```

Figure 3.8: Weak memory update

In such situations, for each possible memory location we just append the contents of `t1` to the contents that are already present in the value set of the given memory location. Finally both in the third and last case the update is ignored.

Algorithm 3.2: analyse_func_bbs(bb)

```
if $(not\ already\_analysed(bb))$ and $all\_parents\_analysed(bb.parents)$ then
    $State_{in} \longleftarrow get\_parents\_state(bb.parents)$
    for all $inst \in bb$ do
        $State_{in} \longleftarrow analyse\_instruction(inst, State_{in})$
    end for
    $bb.analysed \longleftarrow$ true
    for all $son\_bb \in bb.sons$ do
        $analyse\_func\_bbs(son\_bb)$
    end for
else
    $continue$
end if
```

Algorithm 3.3: analyse_instruction(inst, parent_states_in)

```
$State_{in}(inst) \longleftarrow merge\_abs(parent\_states\_in, empty\_state)$
if $is\_call(inst)$ then
  $State_{out}(inst) \longleftarrow handle\_call(inst)$
else
  $State_{out}(inst) \longleftarrow transfer\_function(inst)$
end if
return $State_{out}(inst)$
```

Algorithm 3.4: handle_call(inst)

```
$F \longleftarrow get\_function\_called(inst)$
if $Code\ of\ F\ unavailable$ then
  $State_{out}(inst) \longleftarrow ignore\_call(State_{in}(inst))$
else
  $State_{out}(inst) \longleftarrow analyse\_func(F, State_{in}(inst))$
end if
```

3.5 Modelling the Heap State

To find vulnerabilities on the heap, the author proposes a new heap representation that includes a *status* representing whether a chunk is *allocated* or *freed*. So the previous heap model defined in Figure 3.5 is updated to the one presented in Figure 3.9:

$$\begin{aligned} \langle\text{status}\rangle &::= \textit{allocated} \mid \textit{freed} \\ \langle\text{heap}\rangle &::= \textbf{\textit{He}}\ \langle\text{chunk}\rangle \times \langle\text{status}\rangle \end{aligned}$$

Figure 3.9: Heap status grammar

With the new addition when there is a call to an allocation function it creates a similar value set, $\{$**He**(chunk, allocated), $\{$0x0$\}\}$, where *chunk* is a unique string identifier. On the other hand, when `free()` is called, is is necessary to find all the memory locations of the current memory state that contain the given chunk in its value set and change the chunk *status* property to *freed*. With the new heap model it is simple to detect double-free and use-after-free vulnerabilities. To detect a double-free it is required to check if the value set passed to `free` is a heap object with the status property set to *freed*. Regarding the use-after-free there are two causes that can trigger it, reading from a *freed* heap object or writing into a *freed* heap object. Therefore to detect use-after-free vulnerabilities it is needed to check before executing the *REIL* instructions, `ldm` (load from memory) if the source is a heap object and its status property value is *freed*, and `stm` (store to memory), if the the destination is a heap object and it is marked as *freed*.

4 Solution

Contents

In this chapter we will start by describing the architecture of our solution and our improvements on GUEB. These include:

1. improved detection of use-after-free vulnerabilities which allowed us to increase the recall metric on the datasets
2. support for the *x86-64* architecture that let us analyse not only 32 bit binaries but also 64 bit ones
3. improved memory model which is needed for the detection of heap buffer overflows
4. reduction of memory usage

Afterwards we will focus on extending GUEB so that it also detects heap buffer overflow vulnerabilities.

4.1 Solution architecture

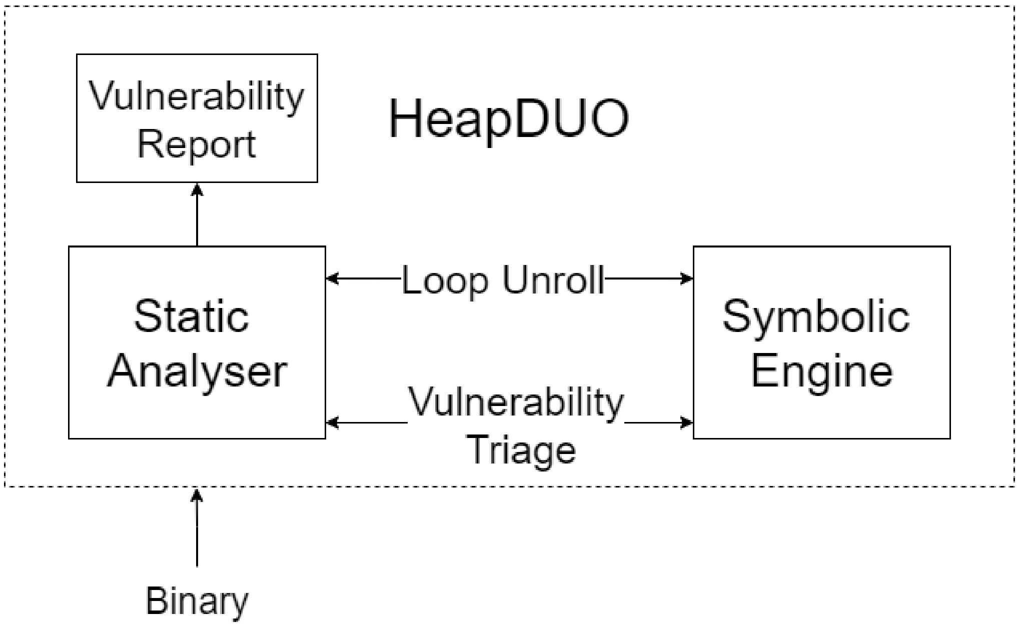


Figure 4.1: Solution architecture

The purpose of this work is to develop a tool capable of automatically detect heap vulnerabilities in a given program.

Architecture: Our solution has 2 major components as seen in Figure 4.1: a static analyser and a symbolic execution engine.

Static analyser: One of the main issues when working with a symbolic execution engine, as mentioned before, is the path explosion problem. Thus, finding heap vulnerabilities can become an unfeasible task when resorting only to symbolic execution. Therefore we will extend the static analyser GUEB [1] not only to find use-after-free and double-free vulnerabilities but also to find heap overflows. The static analysis is performed on the intermediate language *REIL* [30], that can be translated from many different architectures such as x86, PowerPC-32 and ARM-32. This translation is performed by the Binnavi framework [31] that exposes an API [32] which is used to get the basic blocks and their relations, using them to build the control flow graph. The framework extracts the assembly instructions and the control flow graph with the help of the disassembler IDA [29] and BinExport [33]. Along the analysis, HeapDUO

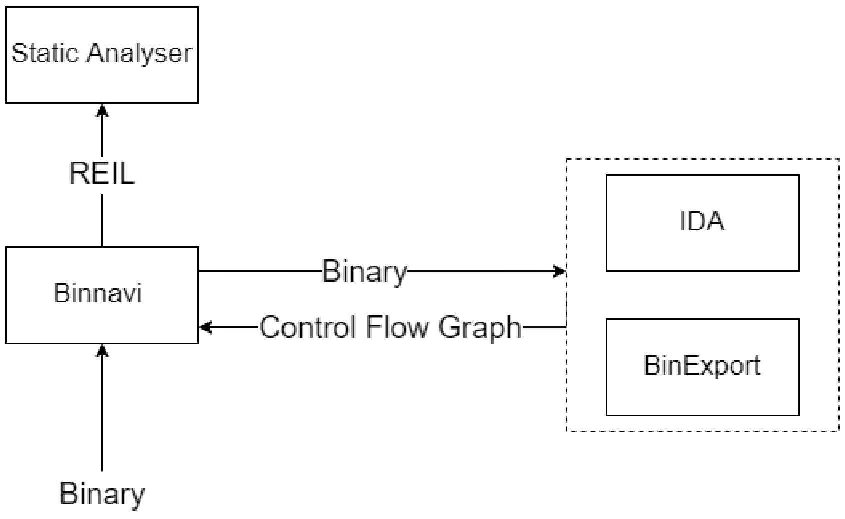


Figure 4.2: Static analysis architecture

may find that certain loops should be unrolled and sends information to the symbolic engine the sequence of instructions that can be followed to reach the loop and the sequence of instructions that must be followed to stay inside it. When it is no longer possible to continue executing the loop, the symbolic engine sends to the static analyser how many times the loop should be unrolled. When the static analysis finishes, it will send to the symbolic engine information regarding the vulnerability that was found and possible sequences of instructions that can be followed to reach it. To detect heap vulnerabilities it is not necessary to have a fine grained implementation of `malloc()/free()`. We will reuse the naive approach developed in GUEB [1], where every call to `malloc()` returns a different memory location.

Symbolic execution engine: The symbolic engine will be responsible for **I)** discover how many times a loop can be unrolled and send it to the static analyser, **II)** detect and inform the static analyser if the information sent by it is indeed a real heap vulnerability. This is similar to the approach taken by KLEE [13], where at each operation that involves heap functions or heap objects it is checked for the existence of a vulnerability such as double-free, use-after-free or heap overflow.

4.2 GUEB Improvements

In this section we present all the improvements made on GUEB.

4.2.1 Use-After-Free Detection

To detect use-after-free vulnerabilities, as stated in Section 3.5, we find memory writes and memory reads where the memory location is a *freed* heap object. In Figure 4.3 there is simple example that demonstrates both of these behaviours, a memory store in line 7 and a memory read in line 8, both from heap objects that are marked as *freed*, which *GUEB* is able to correctly identify as a vulnerability.

A problem arises when the vulnerability does not occur in the available code, but in some shared library. Recall the Algorithm 3.4 `handle_call`, that ignores a function call if the code is not available, therefore *GUEB* is not able to identify the call to `memcpy` on line 10 as a vulnerability resulting in a false negative. Note that we are not trying to find vulnerabilities in the code base of shared libraries. In these cases we want to identify if any of the arguments passed to a function that is not available is a *freed* object. Since we don't have an easy way to recognize how many arguments an external function has based on the *REIL* instructions, we need to have a mapping that associates a function name with the arguments that can possibly be an heap object, we call this mapping `funcArg`.

```
1  int main() {
2
3          char c;
4          char* buffer = (char*) malloc(sizeof(char) * 2);
5          free(buffer);
6
7          buffer[0] = 'A';          // UAF - detected
8          c = buffer[0] + 1;        // UAF - detected
9
10         memcpy(buffer, "A", 1);   // UAF - not detected
11
12         return 0;
13 }
```

Figure 4.3: Use-after-free

For instance `funcArg("memcpy") = {0, 1}`, meaning that both the first and second argument can be heap objects and are a potential source of vulnerabilities, like the one at line 10.With this new mapping we are able to detect use-after-free vulnerabilities in functions that are not available for analysis. In our solution we implemented a total of 20 function mappings for the most commonly used GLIBC [16] functions such as `memset, strcpy/strncpy, strcat/strncat`. The addition of a new mapping is a simple process, the only requirements being the function name and the arguments that should be analysed. The results of this improvement can be seen in the evaluation section (Section 5).

We can imagine a situation where a function with the following signature is called, `function(char**)`, where the argument can be a heap object that is *allocated* and contains an array of heap objects that are *freed*. This approach has the limitation of only checking the argument itself and never its contents, the array of *freed* heap objects. To consider such cases we would need to be structurally aware of the arguments, which would not be difficult to implement but in our work we haven't encountered such functions from GLIBC.

4.2.2 New architecture support

GUEB is only capable of analysing x86 binaries which is a major limitation because x86-64 binaries are predominant nowadays over x86 ones.

With this in mind we wanted to support the analysis for the x86-64 architecture requiring an x86-64 *REIL* translator which was completed by the community in 2020 [34]. We extended this work to support missing instructions such as the `movsxd` instruction, which is emitted when compiling simple statements like `for (int i = 0; i < size; i++) buffer[i] = 'A'`. Furthermore, we also fixed instructions that were incorrectly implemented.

GUEB was built for the x86 architecture, and so we had to adapt it to support x86-64. For instance to get the argument of `malloc`, we need to get the value from the stack if we are working on *x86* but from the register `rdi` if we are working on *x86-64*. Very similarly the return value of a function is set either on the `eax` or the `rax` register according to the architecture. We developed two simple functions, `get_first_arg` and `set_return_value` that have different behaviours depending on the architecture, creating an abstraction not only for these two, but also for all the others that could be supported.

4.2.3 Memory Model

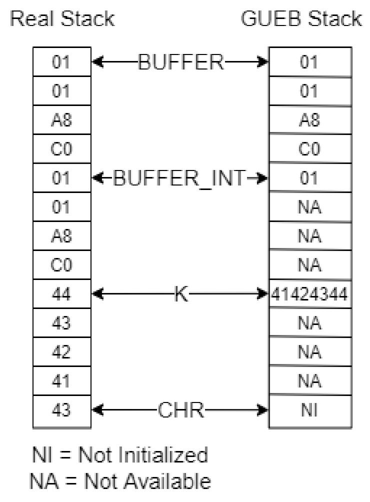
One inconsistency in *GUEB's* memory model is that memory locations are considered to be disjoint. In Figure 4.4, we have an example that explains what we mean by memory locations being disjoint. In Figure 4.4b we have two stacks; to the left a real stack, and to the right an abstract *GUEB* stack to better understand the concept. Both stacks represent the changes done by the code in Figure 4.4a.

The stack values represented by the variable `buffer` are equal. However, when we convert the type `char*` to `int*` and get the first element as an `int` (line 9), *GUEB* does not consider the consecutive memory locations of `buffer` and only gets the first one, thus evaluating `buffer_int` to `0x01`. Another situation that illustrates the same concept is at line 10 where the assignment does not split the value into 4 consecutive memory locations which makes the variable `chr` have a non initialized (NI) value instead of the correct value `0x43` (line 11).

Thus detecting possible heap buffer overflows with the current memory model was not possible

```c
int main() {
    char buffer[4];

    buffer[0] = 0x01;
    buffer[1] = 0x01;
    buffer[2] = 0xa8;
    buffer[3] = 0xc0;

    int buffer_int = *((int*)buffer);
    int k = 0x41424344;
    char chr = ((char*)&k)[1];

    printf("0x%x\n", buffer_int);
    printf("%c\n", chr);
    return 0;
}
```

(a) Source Code



(b) Stack

Figure 4.4: Disjoint memory location

and it was necessary to implement a byte addressed memory model where memory locations are no longer disjoint and where it is possible to reason about buffer boundaries. For instance if we did `strlen((char*)&foo)` we would get 1 instead of 4 due to the contiguous addresses being marked as not available (NA).

To implement a byte addressed memory model we used new mapping functions for the **Globals** and **He** memory region that receive as input an address and return a list of bytes. Since we want our memory locations to be contiguous, we created the function `set_value (memory_location, values, size)` that is responsible for storing `size` bytes into the corresponding `memory_location`. These bytes are the result of applying the function `unpack_values` on `values`, where each *offset* of a *value*, is split into a list of `size` bytes. Note that the argument `size`, recall Section 3.4, is now being used and is taken from the operand size of *REIL* instructions. So for instance, if we want to store {**Constant**, {0x41424344}} in **Globals**(0xa0), we now get: **Globals**(0xa0) → [0x44]; **Globals**(0xa1) → [0x43]; **Globals**(0xa2) → [0x42]; **Globals**(0xa3) → [0x41].

In addition, a heap object can no longer be identified by a *string*, recall Figures 3.5 and 3.9, because its address, like the buffer contents, also has to be stored in contiguous memory. Instead we chose to identify it by a unique heap address similarly to real programs. Therefore our static analysis tool needs to keep track of the next available heap address, which it does by having a variable `next_heap_address`. When we encounter an allocation function like `malloc`, we set our unique identifier to be equal to `next_heap_address` and increment it using the requested size. With this approach we ensure that there are no overlapping heap objects. So we update our heap memory region to include a chunk address instead of a string identifier to:

$$\begin{array}{rcl}
\langle\text{chunk\_addr}\rangle & ::= & \langle\text{addr}\rangle \\
\langle\text{status}\rangle & ::= & \textit{allocated} \mid \textit{freed} \\
\langle\text{heap}\rangle & ::= & \textbf{\textit{He}}\ \langle\text{chunk\_addr}\rangle \times \langle\text{status}\rangle
\end{array}$$

Figure 4.5: Heap addressed grammar

To complete our byte address model we still need to consider memory reads. These are handled by the `get_value (memory_location, size)` function, which essentially gets the address from the memory location and builds `size` lists of bytes. Using `pack_values` we get the original value that was stored in memory and we create a value set. The problem arises from the fact that it is not possible to know if the value that was originally stored is of type **Constant** or of type **He**. To consider this we need a new mapping function `addrMap` that maps an address to a single value set. We use `addrMap` to map a *chunk* address added with each of its *offsets* to the original value set. So when storing a **He** value set in memory we need to insert it in `addrMap`. On the contrary when reading from memory after reconstructing the original number we can tell if it is a **Constant** or a **He** by checking its absence or presence, respectively, in `addrMap`.

Binary programs usually have read/write and read only segments that contain global and static variables, like constants, strings, etc. However, GUEB does not load this data into its memory model. In order to make our model more complete, we import all bytes from these segments so that we can later reason about code like `strlen("AAA")`, where "AAA" comes from the data segment. Also, with this new improvement we no longer need to use the memory location $\textbf{Init}_{\textbf{mem}}$ since we import and load all the data that can possibly be addressed.

With our memory model complete we now need to consider what happens when merging two memory states. Such situation can happen when a basic block has more than one parent so its State_{in} comes from merging its parents $\text{State}_{\text{out}}$. We used the `union (memoryRegionA, memoryRegionB)` function that receives two hash tables representing the memory regions and returns a single list where each entry contains a key (an address) with the corresponding possible bytes from both memory regions. So for instance if we have the following memory regions

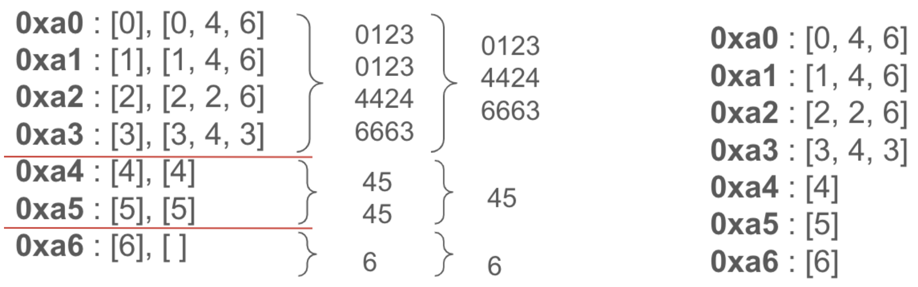
```
    memoryRegionA = {0xa0 = [0]; 0xa1 = [1]; 0xa2 = [2]; 0xa3 = [3]; 0xa4 = [4];
                     0xa5 = [5]; 0xa6 = [6]}
    memoryRegionB = {0xa0 = [0, 4, 6]; 0xa1 = [1, 4, 6]; 0xa2 = [2, 2, 6];
                     0xa3 = [3, 4, 3]; 0xa4 = [4]; 0xa5 = [5]},
```

the result from calling `union` would be equal to

```
    [[0xa0, ([0], [0, 4, 6])], [0xa1, ([1], [1, 4, 6])], [0xa2, ([2], [2, 2, 6])],
     [0xa3, ([3], [3, 4, 3])], [0xa4, ([4], [4])], [0xa5, ([5], [5])], [0xa6 , ([6], [])]].
```

Afterwards we pass this result to `merge_heuristic`, which has a visual representation in Figure 4.6 of its inner workings. In Figure 4.6a, we can see that these addresses are split into 3 sets, {`0xa0, 0xa1, 0xa2, 0xa3`}, {`0xa4, 0xa5`}, {`0x6`}, where each set must have: **I)** consecutive addresses; and **II)** the number of the possible bytes for every address in a given set must be equal. Afterwards, and for every set, we build a sequence of bytes using the possible bytes for each address and remove the duplicated sequences, thus we are left with unique sequences. The final step is to build a new memory region from the sequence of bytes, resulting in the memory region that can be seen in Figure 4.6b. So the final memory region is

{`0xa0 = [0, 4, 6]; 0xa1 = [1, 4, 6]; 0xa2 = [2, 2, 6]; 0xa3 = [3, 4, 3]; 0xa4 = [4]; 0xa5 = [5]; 0xa6 = [6]`}



(a) Memory regions to merge

(b) Memory region result

Figure 4.6: Merge heuristic

The final point left to discuss regarding our improved memory model is the merge of the new mapping `addrMap`. There are 2 cases that need to be considered: the first one is when a given address is only present in one of the memory regions; the other one is when a given address is present in both memory regions. In the first case we just copy the original mapping to the new `addrMap`. The second case results from situations where we have a basic block that receives for instance two memory states where the same heap object is marked as *freed* in only one of them. In these situations we must add to the new mapping the heap object that is *freed*, otherwise possible use-after-free vulnerabilities might not be detected.

4.2.4 Memory Usage

To find possible use-after-frees, GUEB first analyses all nodes of a function, keeping the State_{in} and $\text{State}_{\text{out}}$ data in memory and then revisits every `ldm` and `stm` instruction. Since every assembly instruction can be translated up to 256 *REIL* instructions, and every *REIL* instruction has a State_{in} and a $\text{State}_{\text{out}}$, the memory usage will increase much more than it has to as the analysis proceeds. The only reason we can think of why *GUEB* was implemented this way is that *GUEB* did not have memory usage problems due to its simplistic memory model. However, since our memory model is considerably more complex, it results in more memory necessary to hold each State_{in} and $\text{State}_{\text{out}}$. We improve this by analysing each `ldm` and `stm` instruction for heap vulnerabilities as they come instead of waiting for the whole function to be analysed. This way we can destroy the State_{in} and $\text{State}_{\text{out}}$ data that is longer needed, reducing this way the memory usage.

4.3 Heap buffer Overflow detection

We can think of any heap memory access as a dereference of a heap object base pointer added to a given offset, thus we can easily detect heap overflows by simply checking if the following condition is true

$$\texttt{heap\_object\_base\_pointer + offset >= heap\_object\_base\_pointer + heap\_object\_size},$$

which can be translated to

$$\texttt{offset >= heap\_object\_size}.$$

With the use of the value set analysis algorithm, in every instruction, we are aware if we are dealing with heap objects and we know each heap object base pointer and possible offsets. Note that the heap object base pointer is the *chunk_addr* unique identifier, recall Figure 4.5, and the *offsets* come from the value set itself. At this point our memory model does not keep track of the heap objects size, so once again we need to change our heap memory definition to:

$$\begin{array}{rcl} \langle\text{size}\rangle & ::= & \langle\text{offsets}\rangle \\ \langle\text{status}\rangle & ::= & \textit{allocated} \mid \textit{freed} \\ \langle\text{chunk\_addr}\rangle & ::= & \langle\text{addr}\rangle \\ \langle\text{heap}\rangle & ::= & \textbf{\textit{He}}\ \langle\text{chunk\_addr}\rangle\times\langle\text{status}\rangle\times\langle\text{size}\rangle \end{array}$$

Figure 4.7: Final Heap grammar

Note that *size* holds a list of possible sizes for each heap object. Now our memory model is keeping track of all the information necessary to detect possible read and write heap out of bounds. The function `detect_heap_overflow` detailed in Algorithm 4.1, is called to analyse every `ldm` and `stm` *REIL* instruction, since they are the source of possible heap memory reads and writes. It starts by filtering out all the memory locations that are not part of the heap, `filter_heap_loc`, and for each offset of each heap object it checks whether or not the offset exceeds the `heap_object_size`. Since an heap object can have multiple possible sizes and our goal is to detect heap overflows, we will only consider the smallest one. With this approach we are sure that if there is an out of bounds read/write we will detect it but at the cost of being also a possible source of false positives. Figure 4.8 and Figure 4.9 present an example for each one of these situations. The first one illustrates that by assuming the minimal size to detect overflows we can detect wrong assumptions made regarding the allocated memory size, and the latter presents a false positive originated from considering such assumption. In Figure 4.8 the heap object allocated at line 7 can have a size within the range `[0, 4294967295]`, but in line 8, the programmer assumed that it has a size of at least `13`. With our minimum size assumption we can easily detect that the presented code is vulnerable to a heap out of bounds vulnerability, which happens only when `size <= 12`. On the other hand, the code presented Figure 4.9 does not contain any vulnerability as it allocates a buffer of one of the possible sizes `2,3,4`, and adds a null-byte in the last position given by `size - 1`. Our tool however flags this code pattern as vulnerable because at line 13 we have a heap object dereference where the possible offsets are `{1,2,3}`, and the object size considered is the smallest given by the function `min({2,3,4})`, which is `2`. So when considering `offset = 3`, the condition `offset >= heap_object_size` is satisfied even tough the code is correct.

Algorithm 4.1: detect_heap_overflow(memory_location)

```
overflow_list $\longleftarrow$ []
heap_objects $\longleftarrow$ filter_heap_loc(memory_location)
**for all** heap_object $\in$ heap_objects **do**
    heap_object_size $\longleftarrow$ min(heap_object.size)
    **for all** offset $\in$ heap_object.offsets **do**
        **if** offset $\geq$ heap_object_size **then**
            overflow_list.append(heap_object)
        **end if**
    **end for**
**end for**
**return** overflow_list
```

```c
unsigned int get_size() {
   unsigned int size = 0;
   scanf("%d", &size);
   return size;
}

int main() {
        unsigned int size = get_size();
        char* buffer = (char*) malloc(size);
        buffer[12] = '\0';
        return 0;
}
```

Figure 4.8: Heap out of bound

As we have just seen in Figure 4.8 the `scanf` function was crucial to find the heap overflow since it was the one who dictated the heap object size, thus allowing us to reason about the condition `offset >= heap_object_size`. We can think of other examples where GLIBC functions can be an indirect source of vulnerabilities, for instance, user input can be used as an index when accessing heap objects. `scanf("%d", &idx); heap_object[idx] = 'A'` is one such example. Implementing `scanf` and similar GLIBC functions behaviour will allow us to reason about this types of vulnerabilities.

But GLIBC functions can also be a direct source of heap vulnerabilities, in particular reading or writing more bytes than the size of the heap object is considered to be a heap overflow. In Figure 4.10 we have

```c
unsigned int get_size() {
    if (cond1)
        return 2;
    else if (cond2)
        return 3;
    else
        return 4;
}

int main() {
        unsigned int size = get_size();
        char* buffer = (char*) malloc(size);
        buffer[size - 1] = '\0';
        return 0;
}
```

Figure 4.9: False positive heap out of bound

an example where the function `strcat` is responsible for writing past the allocated memory. `strcat` appends `"BBB"` to `buffer` and adds a null byte terminator which causes the overflow. We model this function and 24 other such as `strcat`, `strcpy`, `read`, `memcpy`, `memset`, `scanf`, allowing us to detect more heap overflows. New functions can be easily added and do not need to be part of the GLIBC.

```c
int main() {

  char* buffer = (char*) malloc(sizeof(char) * 4);
  buffer[0] = 'A';

  strcat(buffer, "BBB");
  return 0;
}
```

Figure 4.10: `strcat` heap out of bound

Until now we have not addressed the possibility of heap overflows happening inside loops which can only be triggered after a specific amount of iterations. Since loops can also be a source of vulnerabilities, in the next sections we will target loops with the goal of identifying when the termination condition is based on user input and also try to unroll loops with the help of symbolic execution.

4.3.1 Taint analysis

Our motivation to add taint tracking to our current memory model is to detect loops where its termination is based on user input. Taint analysis is a method of tracking all variables that are influenced directly or indirectly by user input. In Figure 4.11, both examples contain a loop that writes to the variable `heap_ptr` and can eventually write out of bounds since the termination depends on user input. The main difference between the code in Figures 4.11a and 4.11b is that in the former, the termination condition is tainted before the loop and in the latter, the termination condition gets tainted after one iteration. Our approach to catch both of these code patterns will be to unroll every loop twice and check if the loop's termination condition is tainted.

```c
unsigned int size;
scanf("%u", &size);

for (unsigned int i = 0; i < size; i++)
    heap_ptr[i] = (char) fgetc(stdin);
```

(a)

```c
unsigned int i = 0;
char chr = 0;

while (chr != '\n') {
  chr = (char) fgetc(stdin);
  heap_ptr[i++] = chr;
}
```

(b)

Figure 4.11: Heap loop overflow

The only possible source of taint comes from user input and it is propagated to other temporary registers or memory locations when using operations like the *REIL* `add` or `stm` instructions. When adding two registers, if there are any tainted source registers, then the destination register also gets tainted. The same is true when writing to memory, if the source register is tainted then the memory location also gets tainted.

To support taint analysis we need to change our current definition of value that is defined as *value = base x offsets*. Since we need to keep track if a value is tainted, we change our current definition to *value = base x offsets x taint*, where *taint = true* | *false*. Taking into account that we changed our memory model to no longer store a value set in a given memory location but rather to store bytes, we need an additional mapping function that keeps record of whether a byte at a given addresses is tainted or not. The function `taintMap` receives as input an address and outputs the byte's taintedness. So for instance, if we are storing 8 bytes of {{**Constant**, {1, 2}, false}, {**He**, 0x555555000, status=allocated, size={0x41}, {0x0}, true}} in a given address `addr`, we set `taintMap(addr) = false || true`, which evaluates to true, since one of them is tainted.

From now on our value set is complete and defined as follows:

$$
\begin{array}{rcl}
\langle \text{addr} \rangle & ::= & \mathbb{N} \\
\langle \text{offset} \rangle & ::= & \mathbb{Z} \\
\langle \text{offsets} \rangle & ::= & \langle \text{offset} \rangle \mid \langle \text{offset} \rangle + \langle \text{offsets} \rangle \\
\langle \text{taint} \rangle & ::= & \textit{true} \mid \textit{false} \\
\langle \text{status} \rangle & ::= & \textit{allocated} \mid \textit{freed} \\
\langle \text{chunk\_addr} \rangle & ::= & \langle \text{addr} \rangle \\
\langle \text{size} \rangle & ::= & \langle \text{offsets} \rangle \\
\langle \text{heap} \rangle & ::= & \langle \text{chunk\_addr} \rangle \times \langle \text{status} \rangle \times \langle \text{size} \rangle \\
\langle \text{base} \rangle & ::= & \textbf{\textit{Constant}} \mid \langle \text{heap} \rangle \\
\langle \text{value} \rangle & ::= & \langle \text{base} \rangle \times \langle \text{offsets} \rangle \times \langle \text{taint} \rangle \\
\langle \text{values} \rangle & ::= & \langle \text{value} \rangle \mid \langle \text{value} \rangle + \langle \text{values} \rangle \\
\langle \text{valueSet} \rangle & ::= & \langle \text{values} \rangle \mid \top
\end{array}
$$

Figure 4.12: Final value set grammar

4.3.2 Loop unrolling

In this section, we aim to find potential heap overflows inside loops. As stated before, we already have an algorithm that can recognize loops using the control flow graph of a function and can unroll them a specific number of times. Since we are only interested in finding heap out of bounds reads and writes, our goal is only to unroll loops containing heap accesses. To be unrolled more than the default amount of times, a loop needs to have an instruction where a heap object is dereferenced. So, our approach is to perform the analysis as before, recall Section 3.4, but in the presence of a loop with a heap access, we unroll it and restart the analysis.

In Figure 4.16 we have an example of a heap out of bounds write, where Figure 4.16b is the corresponding control flow graph of Figure 4.16a. In this situation, there is a `buffer` of size `10` and a loop that fills it with user-supplied data. The problem arises from the fact that the loop iterates from `0` to `10`, so there is an off by one error that causes an overflow. Relying solely on the intermediate language *REIL* to understand how many times this loop is going to be executed is a difficult task, so to this end, we rely on a symbolic execution engine [2]. We give the symbolic engine all the possible pairs of instructions that the program can take to reach the entry basic block of a loop and stay inside the loop. A pair of instructions can be seen as the current instruction address and the next instruction address. An example of a pair of instructions from Figure 4.16b is `<0x1189, 0x118d>`. We have two types of instructions pairs `(type A/B, <X, Y>)`: `type A` simply tells the symbolic engine that it can go from `X` to `Y`, where `type B` tells the engine that it should go from `X` to `Y` and if jumping to `Y` is satisfiable then a new loop iteration is possible and the maximum number of iterations is incremented. There are situations where we can have two pairs of instructions like the following `(type A, <X, Y>), (type A, <X, Z>)` where the current instruction is the same but the destinations are different. In Figure 4.16b we have such example, where the instruction `0x11c2` has two possible destinations `0x11c4` and `0x11d4`, thus we have `(type A, <0x11c2, 0x11c4>), (type A, <0x11c2, 0x11d4>)`. The only instruction that can lead to stepping out of the loop is the instruction at address `0x11ef`. So, the instruction pair at this address must be of type B, `(type B, <0x11ef, 0x11ac>)`. When the symbolic engine reaches this instruction, it checks if it is feasible to jump to `0x11ac`. If it is, it continues to explore the next instructions and increments the maximum number of iterations. Note that we exclude all pairs of instructions that do not lead to the target entry loop.

One problem when trying to get the maximum number of iterations of a loop with symbolic execution is that loops can have many conditional branches, leading to state explosion, negatively impacting performance.

4.3.3 Validation of vulnerabilities

This section aims at describing our changes to the symbolic engine AVD [2], in order to detect heap vulnerabilities. Our contributions are:

1. Re-implemented the heap model
2. Added heap safety policies
3. Added a heap heuristic to guide the engine exploration

Since we use path insensitive static analysis, it is expected to have many false positives. Our goal is to triage all results with the help of symbolic execution. The first step is to implement a heap model, for which we will take the same approach as we did when modeling the heap in HeapDUO, where all allocation functions return a fresh memory address. One problem that can arise when using this simple heap approach is that we may run out of memory when we encounter an allocation intensive program because we are never reusing freed memory.

The second step regarding the detection of vulnerabilities is to create heap safety policies, which are functions that verify that a given instruction does not lead to an inconsistent/vulnerable program state. When a safety policy is violated, we halt our program execution and try to generate an input capable of triggering a program crash by solving the current path constraints. Heap overflows and use-after-free vulnerabilities may not necessarily cause a program crash. For instance, an overflow may overwrite data that does not trigger a crash. On the contrary, double-free vulnerabilities usually result in a crash forced by GLIBC since it has mechanisms to detect them in the *tcache* and *fastbin* lists as well as in other kinds of chunks.

Three safety policies need to be implemented: (i) the use-after-free, (ii) the double-free, and (iii) the heap overflow policy. Regarding the first one, in every load/store we check if the source/destination is a freed heap object. If this condition is met, we say that our safety policy was violated and solve the current path restrictions to generate an input. The case of double-free is straightforward to implement by simply checking whether the pointer passed to `free` references a previously freed memory object.

```
1  int main() {
2      char* a = (char*) malloc(sizeof(char) * 5);
3      char* b = (char*) malloc(sizeof(char) * 5);
4
5      a[7] = 'A';
6      return 0;
7  }
```

Figure 4.13: Invalid heap access

Since our heap model returns contiguous heap objects, to detect possible heap overflows we need more than just check whether the accesses belongs to a valid and already allocated memory as for instance in Figure 4.13 we have a heap out of bounds where the access belongs to a valid allocated memory. Therefore, and to avoid missing these patterns, we need to know in every load/store the original heap object base pointer so that we can reason whether there is an invalid access. To this end we added a base pointer to the abstract data type, `ADT`, Figure 4.14. This `ADT`, only has its base pointer initialized when an allocation function is called. So all the allocation functions return `ADT(new_heap_addr, base_pointer = new_heap_addr)`. And as a result in Figure 4.13 at line 5 when we are writing out of bounds, we can access the `base_pointer`, that is for example `0x555555559000`, get the memory size that is `5`, and reason that `0x555555559007 - base_pointer = 7`, which is larger than the memory size concluding that we are in the presence of an heap overflow. Note that there are situations where the user is able to control the arguments of an allocation function, which results in being able to control the size of the requested memory. In these cases the size is a symbolic value and it must be converted into a real value. As we did before we will try to minimize it based on the current restrictions over it, so we detect heap overflows like the one in Figure 4.8.

```
class ADT:
    def __init__(self, val, base_pointer = None):
        self.val = val

        # used to keep track of heap OOB read/write,
        # this value is never changed after initialization
        self.base_pointer = base_pointer
```

Figure 4.14: Abstract data type

To this point AVD is now able to detect heap overflows, double-free and use-after-free vulnerabilities, so the final step is to detail the triage process describing the information sent from HeapDUO to AVD.

This information has a specific structure and format. In the case we are in the presence of an overflow vulnerability we have the following structure (<CONTROL FLOW GRAPH>, "OVERFLOW", <ALLOCATION EVENT>, <OVERFLOW EVENT>), otherwise we have (<CONTROL FLOW GRAPH>, "DOUBLE FREE"/"USE AFTER FREE", <ALLOCATION EVENT>, <FREE EVENT>, <VULNERABILITY EVENT>). Both have enough details for the symbolic engine to start the analysis and to reach each of the events and declare if it is in fact a vulnerability. Regarding the format of the <CONTROL FLOW GRAPH>, it contains only the necessary information to build a minimal control flow graph to reach all of the previous events when traversing it. The <ALLOCATION/FREE/VULNERABILITY EVENT> has the information regarding where the allocation, free or vulnerability occurs, such as the basic block address, the instruction address, and the call stack required to reach it. A call stack is a dynamic structure used by HeapDUO that works like a stack, where we insert on top the instruction address that called the function that is going to be analysed, and when we finish analysing a function we pop the first address from the it. It is essential that each event contains a call stack and we illustrate such reason with the example in Figure 4.15. Without a call stack we would only have the information that the overflow happened in the instruction at line 4, therefore when analysing the function that is called at line 12 none of the heap safety policies would be violated, meaning that the engine would report it as a false positive. Yet the vulnerability only occurs when the analysis calls fill_buffer at line 13, thus the importance of a call stack for each event.

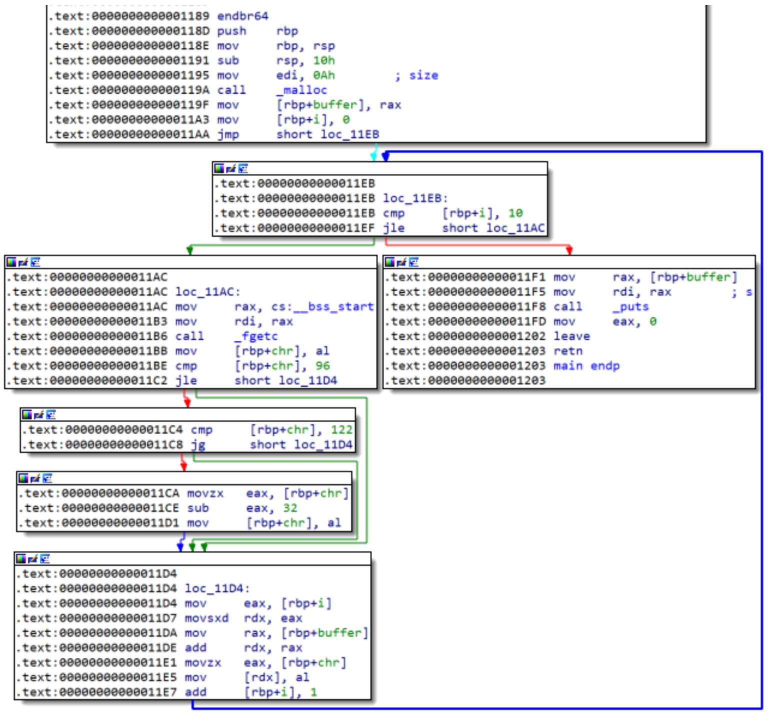
```
1  void fill_buffer(char* buffer) {
2    #define SIZE 10
3    for (int i = 0; i < SIZE; i++)
4      buffer[i] = 'A';
5  }
6
7  int main() {
8
9    char* buffer1 = (char*) malloc(sizeof(char) * 20);
10   char* buffer2 = (char*) malloc(sizeof(char) * 1);
11
12   fill_buffer(buffer1);
13   fill_buffer(buffer2); // overflow
14
15   return 0;
16 }
```

Figure 4.15: fill_buffer overflow

As stated before, a new instance of the symbolic execution engine is forked at each conditional branch, which leads to the known state explosion, therefore we try to guide our engine reaching each of these events. AVD already provides a template for this with a function prior(mem1, mem2) that lets us choose which memory, mem1 or mem2 we want to prioritize. The high level idea is to choose the memory that is closer to the next event. Note that to trigger a safety policy violation, the symbolic engine needs to reach each of the events imported from HeapDUO in order. So for instance in the case of a use-after-free vulnerability it is required to reach the allocation event first, then the free event and finally the vulnerability itself. To find which memory is closer to reaching the next event, we use a breadth first search in the control flow graph to count the distance from the current basic block to the basic block where the target event occurs. In case the distances are equal for both memories we randomly select one of them. Also if when AVD, finds memories that cannot reach the next target event it stops analysing them. Finally for each vulnerability reported by HeapDUO, the symbolic engine informs it whether it is a real vulnerability.

```c
#define SIZE 10
int main() {
  char chr;
  char* buffer;
  buffer = (char*) malloc(sizeof(char) * SIZE);

  for (int i = 0; i <= SIZE; i++) {
    chr = (char) fgetc(stdin);

    if (chr >= 'a' && chr <= 'z')
      chr = chr - ('a' - 'A');

    buffer[i] = chr;
  }
  puts(buffer);

  return 0;
}
```

(a) Source code



```
.text:0000000000001189 endbr64
.text:000000000000118D push    rbp
.text:000000000000118E mov     rbp, rsp
.text:0000000000001191 sub     rsp, 10h
.text:0000000000001195 mov     edi, 0Ah        ; size
.text:000000000000119A call    _malloc
.text:000000000000119F mov     [rbp+buffer], rax
.text:00000000000011A3 mov     [rbp+i], 0
.text:00000000000011AA jmp     short loc_11EB
```

```
.text:00000000000011EB
.text:00000000000011EB loc_11EB:
.text:00000000000011EB cmp     [rbp+i], 10
.text:00000000000011EF jle     short loc_11AC
```

```
.text:00000000000011AC
.text:00000000000011AC loc_11AC:
.text:00000000000011AC mov     rax, cs:__bss_start
.text:00000000000011B3 mov     rdi, rax
.text:00000000000011B6 call    _fgetc
.text:00000000000011BB mov     [rbp+chr], al
.text:00000000000011BE cmp     [rbp+chr], 96
.text:00000000000011C2 jle     short loc_11D4
```

```
.text:00000000000011F1 mov     rax, [rbp+buffer]
.text:00000000000011F5 mov     rdi, rax        ; s
.text:00000000000011F8 call    _puts
.text:00000000000011FD mov     eax, 0
.text:0000000000001202 leave
.text:0000000000001203 retn
.text:0000000000001203 main endp
.text:0000000000001203
```

```
.text:00000000000011C4 cmp     [rbp+chr], 122
.text:00000000000011C8 jg      short loc_11D4
```

```
.text:00000000000011CA movzx   eax, [rbp+chr]
.text:00000000000011CE sub     eax, 32
.text:00000000000011D1 mov     [rbp+chr], al
```

```
.text:00000000000011D4
.text:00000000000011D4 loc_11D4:
.text:00000000000011D4 mov     eax, [rbp+i]
.text:00000000000011D7 movsxd  rdx, eax
.text:00000000000011DA mov     rax, [rbp+buffer]
.text:00000000000011DE add     rdx, rax
.text:00000000000011E1 movzx   eax, [rbp+chr]
.text:00000000000011E5 mov     [rdx], al
.text:00000000000011E7 add     [rbp+i], 1
```

(b) Control flow graph

Figure 4.16: Heap out of bound write

5 Evaluation

Contents

In this chapter we present our tool performance where we focus on showing that the tool developed is able to:

1. detect heap vulnerabilities in a given application;
2. triage the false positives that are generated from the static analysis;
3. work on both *x86* and *x86-64* architectures.

In order to assess HeapDUO we relied on two different datasets: the Juliet [35] dataset "(...) created by the National Security Agency's (NSA) Center for Assured Software (CAS) and developed specifically for assessing the capabilities of static analysis tools." and on the CodeQL dataset [36] "CodeQL is the analysis engine used by developers to automate security checks, and by security researchers to perform variant analysis".

The metrics considered on both datasets are:

1. the time it takes to analyse all the binaries for each vulnerability;
2. recall — the fraction between the number of the detected vulnerabilities and the number of vulnerable tests;
3. precision — fraction between the number of false positives and the number of tests that were not vulnerable.

For each dataset we were able to generate vulnerable and non-vulnerable binaries for the two supported architectures, so we are certain about the precision and the recall of our tool. We also compared our performance both in terms of the number of detected vulnerabilities and the time it took to analyse the binaries with GUEB and AFL++ [3]. Finally, we reanalysed CVE-2015-5221, CVE-2015-8871 and CVE-2016-3177, CVE's [37] found by GUEB, in order to test whether we are still capable of finding these same vulnerabilities; we also present our tool capability of finding real world vulnerabilities by detecting CVE-2021-32614 [4], a heap out of bounds read, that was reported in 2021.

In the following two sections we present our metrics on both the Juliet and the CodeQL datasets, where we separated them in terms of the binaries containing vulnerabilities and safe binaries. We evaluate the tools in terms of success in vulnerability detection #detected and false positives (#FP) and in terms of execution total time (TT), average time (AVG) and the average time taken by the static analyser (SA AVG). Note that since AFL++ feeds mutated inputs to the target program indefinitely even when it finds a crash, we had to change its source in order to stop execution right after finding the first program crash. Additionally we set a timeout of `3 x 60 = 180` seconds from which we terminate AFL++ execution and declare the vulnerability as not found.

5.1 Juliet dataset

5.1.1 Dataset with vulnerabilities

Dataset		HeapDUO		AFL++	
CWE 416/415/122	#tests	#detected	recall	#detected	recall
Use After Free	118	118	**100%**	0	0%
Double Free	190	190	**100%**	187	98.4%
Heap Overflow	1178	1178	**100%**	54	4.6%

Table 5.1: Vulnerability detection in 64 bit dataset with vulnerabilities

	HeapDUO			AFL++	
CWE 416/415/122	TT (sec)	AVG (sec)	SA AVG (sec)	TT (sec)	AVG (sec)
Use After Free	**3469.66**	**29.40**	**1.96**	21240.00	180.00
Double Free	2218.75	11.67	0.04	**571.75**	**3.00**
Heap Overflow	**34307.55**	**29.12**	**0.68**	202328.72	171.75

Table 5.2: Time taken to analyze Juliet 64 bit dataset with vulnerabilities

Table 5.1 shows the number of detected vulnerabilities while analysing *x86-64* binaries. We can see that HeapDUO was able to correctly identify all the considered vulnerabilities namely use-after-free, double-free and heap overflow. On the contrary AFL++ performed poorly both in the binaries containing both use-after-free and heap overflow vulnerabilities ending up with a recall of `0%` and `4.6%` respectively. We can think of two reasons why AFL++ performance was unsatisfactory:

1. The Juliet binaries had close to none heap interactions after the vulnerability occurred, which made it impossible to enter in a corrupt state

2. Since our thesis works at machine code level, no source code information is allowed to analyse the binaries, therefore AFL++ could not instrument the binary with ASan [38], a memory detector for C/C++ that is able to find use-after-free, double-free and heap overflows.

In the case of double-free vulnerability, AFL++ had a recall of `98.4%`, since GLIBC security checks are able to identify these situations and abort execution. Regarding the time performance, Table 5.2, shows that HeapDUO proved to be faster in the detection of use-after-free and heap overflows while the detection of double-free's was three times slower. GUEB was not considered in these tests because it does not support the *x86-64* architecture.

In Table 5.3 we have the results of analysing *x86* vulnerable binaries, and here we can see that our improved use-after-free detection increased the recall metric of GUEB from `64.4%` to `100%` meaning that we were able to detect vulnerabilities that GUEB could not. We were also able to detected more double-free vulnerabilities than GUEB. AFL++ had a similar performance comparing to the results in Table 5.1 due to the reasons previously explained.

Dataset		HeapDUO		GUEB		AFL++	
CWE 416/415/122	#tests	#detected	recall	#detected	recall	#detected	recall
Use After Free	118	118	**100%**	76	64.4%	0	0 %
Double Free	190	190	**100%**	185	97.3%	186	97.9 %
Heap Overflow	1254	1254	**100%**	-	-	55	4.4 %

Table 5.3: Vulnerability detection in Juliet 32 bit dataset with vulnerabilities

Dataset	HeapDUO			GUEB		AFL++	
CWE 416/415/122	TT (sec)	AVG (sec)	SA AVG (sec)	TT (sec)	AVG (sec)	TT (sec)	AVG (sec)
Use After Free	2289.08	19.39	1.88	**3.37**	**0.02**	21240.00	180
Double Free	1147.62	6.04	0.11	**4.28**	**0.02**	735.54	3.87
Heap Overflow	**17911.73**	**14.28**	**0.80**	-	-	215824.83	172.10

Table 5.4: Time taken to analyze Juliet 32 bit dataset with vulnerabilities

As can be seen in Table 5.1 and in Table 5.3 there is a difference between the number of analysed heap overflow binaries, 1178 and 1254, respectively. This difference comes from the fact that some vulnerabilities could only be triggered in *x86* due to the size of pointers in *x86* being 4 bytes as opposed to 8 bytes in *x86-64*.

Analysing Table 5.4 we can see a significant difference between the time performance of GUEB and HeapDUO. The two main reasons for the observed decrease in HeapDUO performance are

1. the merge operation between memory states takes additional time due to the more complex memory model;
2. the addition of a symbolic engine that triages the vulnerabilities found by HeapDUO.

5.1.2 Dataset without vulnerabilities

Regarding the analysis of the dataset without vulnerabilities, Table 5.5 and Table 5.6 reveal that both HeapDUO and GUEB reported 0 false positives. It should be mentioned that the static analyser of HeapDUO reported false positives due to loop unrolls but were discarded by the symbolic engine.

Dataset	HeapDUO			
CWE 416/415/122	#tests	#FP discarded	#FP	precision
Use After Free	118	31	0	100%
Double Free	190	5	0	100%
Heap Overflow	1178	269	0	100%

Table 5.5: Vulnerability detection in Juliet 64 bit dataset without vulnerabilities

Dataset		HeapDUO			GUEB	
CWE 416/415/122	#tests	#FP discarded	#FP	precision	#FP	precision
Use After Free	118	90	0	100%	0	100%
Double Free	190	5	0	100%	0	100%
Heap Overflow	1178	350	0	100%	-	-

Table 5.6: Vulnerability detection in Juliet 32 bit dataset without vulnerabilities

5.2 CodeQL dataset

The CodeQL dataset contained 19 handmade programs vulnerable to heap buffer overflows and 12 programs without any kind of vulnerabilities. For this reason only HeapDUO and AFL++ were taken into consideration in the following results.

5.2.1 Dataset with vulnerabilities

Dataset		HeapDUO		AFL++	
CWE 122	#tests	#detected	recall	#detected	recall
Heap Overflow	19	19	**100%**	0	0%

Table 5.7: Vulnerability detection in CodeQL 64 bit dataset with vulnerabilities

Dataset	HeapDUO			AFL++	
CWE 122	TT (sec)	AVG (sec)	SA AVG (sec)	TT (sec)	AVG (sec)
Heap Overflow	85.87	**4.51**	0.04	3420.00	180.00

Table 5.8: Time taken to analyse CodeQL 64 bit dataset with vulnerabilities

Dataset		HeapDUO		AFL++	
CWE 122	#tests	#detected	recall	#detected	recall
Heap Overflow	19	19	**100%**	0	0%

Table 5.9: Vulnerability detection in CodeQL 32 bit dataset with vulnerabilities

Dataset	HeapDUO			AFL++	
CWE 122	TT (sec)	AVG (sec)	SA AVG (sec)	TT (sec)	AVG (sec)
Heap Overflow	85.56	**4.50**	0.08	3420.00	180.00

Table 5.10: Time taken to analyse CodeQL 32 bit dataset with vulnerabilities

Tables 5.7 and 5.9 show that HeapDUO was able to detect all the 19 heap overflows independently of the architecture being analysed. On the contrary AFL++ did not detect any vulnerability due to the same reasons explained before. Regarding the time taken to analyse each program HeapDUO was faster than AFL++ with an average time of 4.51 seconds on *x86-64* and 4.50 seconds on *x86* as can be seen in Tables 5.8 and 5.10, respectively.

5.2.2 Dataset without vulnerabilities

Dataset	HeapDUO			
CWE 122	#tests	#FP discarded	#FP	precision
Heap Overflow	12	0	0	100%

Table 5.11: Vulnerability detection in CodeQL 64 bit dataset without vulnerabilities

Dataset	HeapDUO			
CWE 122	#tests	#FP discarded	#FP	precision
Heap Overflow	12	0	0	100%

Table 5.12: Vulnerability detection in CodeQL 32 bit dataset without vulnerabilities

Regarding the CodeQL dataset without vulnerabilities, Tables 5.11 and 5.12, show that the static analyser did not generate any false positives for any of the 12 tests, so the symbolic engine was not necessary in the triage process. It should be noted that the static analyser generated 0 false positives since the tests were very specific and simple.

5.3 Real World Vulnerabilities

GUEB showed its success by finding three vulnerabilities in open source software which were assigned an unique CVE, CVE-2015-5221 [39] (Jasper-JPEG-200), CVE-2015-8871 [40] (openjpeg) and CVE-2016-3177 [41] (giflib). In order to assess if HeapDUO was still able to find these same heap vulnerabilities we tested HeapDUO on these three projects but were only able to find two of them, CVE-2015-5221 and CVE-2016-3177. To this end we checked if the open sourced GUEB was able to find CVE-2015-8871, which to our surprise it did not. One possible explanation for this situation is that perhaps the source code of the project, openjpeg, was changed in order to analyse only possible critical functions. Therefore it is reasonable that HeapDUO did not find the last vulnerability since the available version of GUEB could not also.

In order to show the success of our improvements regarding heap overflow detection we searched for recent heap overflow vulnerabilities reported on open source projects, where we found CVE-2021-32614 [4] on *dmg2img* [5], a tool which allows converting Apple compressed dmg archives to standard (hfsplus) image desk files. Our tool reports that a heap out of bounds read happens at the instruction that calls `memcpy()`. Analysing the found vulnerability reported by HeapDUO, we realize that it is assumed that the buffer from which bytes are copied from has 204 bytes, but in reality it can have less which leads to an out of bounds read.

6

Conclusion

Contents

6.1 Conclusions

In this work, we developed HeapDUO, a tool capable of finding heap vulnerabilities such as double-frees, use-after-frees, and heap overflows, by performing the analysis only on binaries without any access to source code.

Our tool has two major components, a static analyser based on GUEB [1], and a symbolic engine AVD [2]. Our static analyser performs value set analysis on all REIL instructions of the recovered control flow graph given by the Binnavi Framework. In order to detect heap vulnerabilities, we used a simple heap model where each allocation returns a different memory location. Later on the analysis, these locations can be marked as allocated or freed. Consequently, to find double-frees we check if an already freed location is passed to `free()`; to find use-after-frees we find memory reads or memory writes where the memory location is a freed heap object; and finally to detect heap overflows we look for memory reads and writes on heap objects and check if they are accessed beyond their allocated size. Our symbolic engine is responsible for finding how many times a loop can be executed so that the static analyser can unroll it and continue its analysis. The symbolic engine is also responsible for the triage of vulnerabilities found by the static analyser.

In this thesis we made several improvements both on GUEB and on the symbolic engine AVD. Regarding the static analyser which is based on GUEB, we improved the detection of use-after-free vulnerabilities by also considering the cases when the vulnerability occurs in an external function, added support for the x86-64 architecture, improved the existing memory model to reason about possible heap buffer overflows, improved the memory usage, and added the ability to detect heap overflows. To enhance the detection of heap overflows in loops, we combined our static analyser and the symbolic engine AVD to help determine the number of times a loop can be executed. Additionally, we used AVD to triage vulnerabilities reported by the static analyser, reducing the number of false positives.

Finally, we evaluated HeapDUO by comparing the number of detected vulnerabilities between HeapDUO, GUEB, and AFL++ and the time taken to analyse them in both the Juliet and CodeQL datasets. In terms of effectiveness, HeapDUO managed to find all vulnerabilities present in both datasets. As for timing performance, HeapDUO revealed to be slower than GUEB and AFL++. We also assessed HeapDUO's ability to find vulnerabilities in open-source software by detecting previously found vulnerabilities by GUEB and by detecting a heap out of bounds read, in a open-source project, dmg2img.

6.2 Future work

Our work is dependent on the Binnavi framework since it is the one responsible for translating x86 and x86-64 assembly code to REIL. Since Binnavi is no longer actively maintained and lacks full support for floating point operations on both the x86 and x86-64, one possible future direction is to move from

REIL to another intermediate language. As a result we would have to rewrite some parts of HeapDUO, namely the intermediate language parsing.

Additionally, we improved GUEB's use-after-free detection by adding a mapping that associates an external function name with its arguments that can be an heap object. Even tough this solution increased the detection of use-after-free vulnerabilities on the datasets, it is not complete since it does not take into consideration functions that can receive as an argument a structure containing a heap object. So, to make our analysis more complete when finding use-after-free vulnerabilities we could take all the GLIBC function signatures to be structurally aware of their arguments.

Finally, we could extend HeapDUO with summaries of more external functions to complement the 20 already implemented.

Bibliography

[1] J. Feist, “Finding the needle in the heap : combining binary analysis techniques to trigger use-after-free,” Theses, Université Grenoble Alpes, Mar. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01681707

[2] N. Sabino, “Automatic vulnerability detection: Using compressed execution traces to guide symbolic execution,” Thesis, Nov. 2019.

[3] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[4] “CVE-2021-32614 Details,” https://nvd.nist.gov/vuln/detail/CVE-2021-32614, accessed: 2021-11-01.

[5] “dmg2img,” https://github.com/Lekensteyn/dmg2img, accessed: 2021-11-01.

[6] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, “The matter of heartbleed,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 475–488. [Online]. Available: https://doi.org/10.1145/2663716.2663755

[7] “Mitre, top 25 most dangerous software errors,” https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, accessed: 2021-11-01.

[8] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[9] “Pwn2own 2021 results,” https://www.zerodayinitiative.com/blog/2021/4/2/pwn2own-2021-schedule-and-live-results, accessed: 2021-11-01.

[10] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, no. 2, p. 82–90, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408795

[11] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[12] T. Avgerinos, S. Cha, B. Hao, and D. Brumley, “Aeg: automatic exploit generation. in proc. of the network and distributed system security symposium,” 2011.

[13] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. USA: USENIX Association, 2008, p. 209–224.

[14] P. Godefroid, N. Klarlund, and K. Sen, “Dart: Directed automated random testing,” *SIGPLAN Not.*, vol. 40, no. 6, p. 213–223, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065036

[15] K. Sen, D. Marinov, and G. Agha, “Cute: A concolic unit testing engine for c,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, p. 263–272, Sep. 2005. [Online]. Available: https://doi.org/10.1145/1095430.1081750

[16] “Free software foundation. the gnu c library,” https://www.gnu.org/software/libc/, accessed: 2021-11-01.

[17] “Free software foundation. malloc(3) - linux manual page,” https://man7.org/linux/man-pages/man3/malloc.3.html, accessed: 2021-11-01.

[18] “Free software foundation. mallocinternals - glibc wiki,” https://sourceware.org/glibc/wiki/MallocInternals, accessed: 2021-11-01.

[19] “Doug lea. a memory allocator,” http://gee.cs.oswego.edu/dl/html/malloc.html, accessed: 2021-11-01.

[20] “osip. open source session initiation protocol,” https://www.gnu.org/software/osip/osip.html, accessed: 2021-11-01.

[21] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446.

[22] M. Vittek, P. Borovansky, and P.-E. Moreau, “A simple generic library for c,” in *Proceedings of the 9th International Conference on Reuse of Off-the-Shelf Components*, ser. ICSR’06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 423–426. [Online]. Available: https://doi.org/10.1007/11763864_38

[23] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[24] “Darpa. darpa announces cyber grand challenge,” https://www.darpa.mil/news-events/2013-10-22, accessed: 2021-11-01.

[25] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Heaphopper: Bringing bounded model checking to heap implementation security,” in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC’18. USA: USENIX Association, 2018, p. 99–116.

[26] I. Yun, D. Kapil, and T. Kim, “Automatic techniques to systematically discover new heap exploitation primitives,” 03 2019.

[27] “Zalewski american fuzzy lop,” http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: 2021-11-01.

[28] G. Balakrishnan and T. Reps, “Wysinwyx: What you see is not what you execute,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, Aug. 2010. [Online]. Available: https://doi.org/10.1145/1749608.1749612

[29] “Hex rays,” https://hex-rays.com/, accessed: 2021-11-01.

[30] T. Dullien and S. Porst, “Reil: A platform-independent intermediate representation of disassembled code for static code analysis,” 01 2009.

[31] “Binnavi,” https://github.com/google/binnavi, accessed: 2021-11-01.

[32] “Binnavi api,” https://www.zynamics.com/binnavi/manual/API/html/namespaces.html, accessed: 2021-11-01.

[33] “Binexport,” https://github.com/google/binexport, accessed: 2021-11-01.

[34] “REIL x64 support,” https://github.com/google/binnavi/pull/120, accessed: 2021-11-01.

[35] “Juliet test suite,” https://samate.nist.gov/SRD/resources/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf, accessed: 2021-11-01.

[36] “CodeQL,” https://github.com/github/codeql/tree/1c78c792ffc68a59d1d1d5b301f72adfda13562f/cpp/ql/test/query-tests/Critical/OverflowCalculated, accessed: 2021-11-01.

[37] “CVE,” https://www.redhat.com/en/topics/security/what-is-cve, accessed: 2021-11-01.

[38] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *USENIX ATC 2012*, 2012. [Online]. Available: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[39] “CVE-2015-5221 Details,” https://nvd.nist.gov/vuln/detail/CVE-2015-5221, accessed: 2021-11-01.

[40] “CVE-2015-8871 Details,” https://nvd.nist.gov/vuln/detail/CVE-2015-8871, accessed: 2021-11-01.

[41] “CVE-2016-3177 Details,” https://nvd.nist.gov/vuln/detail/CVE-2016-3177, accessed: 2021-11-01.