HeapDUO: Finding Heap Vulnerabilities in Binary Code

Jorge Cardoso Martins
jorge.cardoso.martins@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

Oct 2021

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, *D*ouble-frees, *U*se-after-frees and Heap *O*verflows. It also relates to the fact that it is composed by two major components, a static analyser and a symbolic engine. We extended GUEB [15], 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 [18] 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++ [16], 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, in the open-source project dmg2img.

Keywords: Static Analysis, Symbolic Execution, Heap Vulnerabilities, Automatic Detection,

1. Introduction

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 call it a vulnerability.

Security vulnerabilities can have huge impacts not only for the customers but also to the reputation of companies. For instance Heartbleed [13] was a vulnerability found in OpenSSL cryptography 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 [3], 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++ [21] 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 writes past the previously allocated memory. To clearly understand how dangerous these are, in Pwn2Own 2021 [4], 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 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 where in fact there are not. To filter and verify 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, and a symbolic engine, AVD. 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 new 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 tested HeapDUO's capabilities of detecting previous bugs found by GUEB and finally it detected a heap out bound read, already known, in an open-source project.

1.4. Outline

This thesis is divided as follows: In Section 2 we provide a background for the work developed, where we describe symbolic execution and the heap. In Section 4 we detail an open-source project that is capable of finding some heap vulnerabilities and in which our work is based on. In Section 5 we describe our solution architecture along with our implementation choices. Section 6 presents the evaluation of our work in two datasets, the Juliet and CodeQL dataset, and also the detection of vulnerabilities in open-source projects. Finally, Section 7 concludes our thesis with future work.

2. Background

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.

```c
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 1: C program example

For example, symbolic execution of the program in Figure 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,

$PC_0 = \{true \land (x_0 + y_0 > 32)\}$, and a new path constraint, PC_1, is created with the symbolic expression negated, $PC_1 = \{true \land \neg(x_0 + y_0 > 32)\}$ 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 $\{x = 33, y = 0\}$.

2.2. 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.

2.3. 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 [7], [5] 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 [5], 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 [9] is able to model symbolic files, where these are supported through a basic symbolic file system for each execution state. AEG [5] models symbolic files in a similar approach to KLEE, symbolic sockets, environment variables, library function calls and over 70 system calls. DART [17] and CUTE [19] on the other hand, execute external calls by using concrete arguments, therefore some paths will not be explored.

2.4. Heap

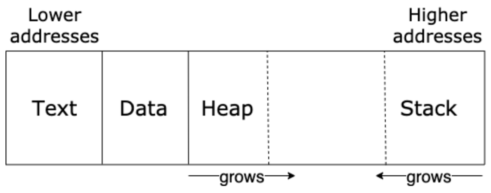

Figure 2: User space program memory regions

A program consists of 4 key memory regions depicted in Figure 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.

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++.

3. Related Work

3.1. Symbolic execution

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

KLEE 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 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*, 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 [8], 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 "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.

3.2. Symbolic Execution Applied to Security

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

In AEG, 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, were applied to MAYHEM [11] the winner of DARPA Cyber Grand Challenge. The vulnerabilities considered by MAYHEM were the same as in AEG.

HeapHopper [14] 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 [23] 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.

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++, 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.

4. GUEB

In this section we describe Graphs of Use-After-Free Extracted from Binary (GUEB) from Josselin Feist [15], 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.

4.1. 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 [6]. 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.

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) $\mathbf{Init_{reg}}$, $\mathbf{Init_{mem}}$: 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:

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

Figure 3: 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 4.

$$\begin{array}{rcl}
\langle\text{offsets}\rangle & ::= & \langle\text{offset}\rangle \ \mid\ \langle\text{offset}\rangle\ +\ \langle\text{offsets}\rangle \\
\langle\text{base}\rangle & ::= & \boldsymbol{Constant}\ \mid\ \langle\text{heap}\rangle \\
 & & \mid\ \boldsymbol{Init_{reg}}\ \langle\text{init\_reg\_name}\rangle\ \mid \\
 & & \mid\ \boldsymbol{Init_{mem}}\ \langle\text{init\_mem\_name}\rangle\ \mid\ \top \\
\langle\text{value}\rangle & ::= & \langle\text{base}\rangle\times\langle\text{offsets}\rangle \\
\langle\text{values}\rangle & ::= & \langle\text{value}\rangle\ \mid\ \langle\text{value}\rangle\ +\ \langle\text{values}\rangle \\
\langle\text{valueSet}\rangle & ::= & \langle\text{values}\rangle\ \mid\ \top
\end{array}$$

Figure 4: Value set grammar

4.2. GUEB VSA algorithm

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.

5. Implementation

In this section 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. 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: 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 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* [12], that can be translated from many different architectures such as x86, PowerPC-32 and ARM-32. This translation is performed by the Binnavi framework that exposes an API 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[2] and BinExport [1].

Along the analysis, HeapDUO 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, 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, 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.

5.1. Use-After-Free Detection

To detect use-after-free vulnerabilities, we find memory writes and memory reads where the memory location is a *freed* heap object.

A problem arises when the vulnerability does not occur in the available code, but in some shared library. 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 use a mapping function, `funcArg,` that associates a function name with the arguments that can possibly be an heap object.

5.2. 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. 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.

5.3. Memory Model

One inconsistency in *GUEB's* memory model is that memory locations are considered to be disjoint. Thus detecting possible heap buffer overflows with the current memory model was not possible 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.

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` is taken from the operand size of the REIL instructions.

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 $\mathbf{Init_{mem}}$ since we import and load all the data that can possibly be addressed.

5.4. 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. 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.

5.5. Heap 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: `heap_object_base_pointer + offset >= heap_object_base_pointer + heap_object_size`, which can be translated to `offset >= heap_object_size`.

At this point our memory model does not keep track of the heap objects size, so we need change our heap memory definition to:

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

Figure 5: Final Heap grammar

With our memory model complete we present our algorithm `detect_heap_overflow` (Algorithm 1).

Algorithm 1 detect_heap_overflow(memLoc)

```
overflow_list ⟵ []
heap_objects ⟵ filter_heap_loc(memoryLoc)
for all heap_object ∈ heap_objects do
    heap_object_size ⟵ min(heap_object.size)
    for all offset ∈ heap_object.offsets do
        if offset ≥ heap_object_size then
            overflow_list.append(heap_object)
        end if
    end for
end for
return overflow_list
```

5.6. Loop unrolling

We have an algorithm from GUEB 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. Relying solely on the intermediate language *REIL* to understand how many times a loop is going to be executed is a difficult task, so to this end, we rely on a symbolic execution engine. 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, where a pair of instructions can be seen as the current instruction address and the next instruction address.

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.

5.7. Validation of vulnerabilities

This section aims at describing our changes to the symbolic engine AVD, 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. 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.

6. Results

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 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 *"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++ [16]. Finally, we reanalysed CVE-2015-5221, CVE-2015-8871 and CVE-2016-3177, CVE's 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, 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.

6.1. Juliet dataset

Table 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

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 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 2: Time taken to analyze Juliet 64 bit dataset with vulnerabilities

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 [20], 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 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.

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 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 4: Time taken to analyze Juliet 32 bit dataset with vulnerabilities

In Table 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 1 due to the reasons previously explained.

As can be seen in Table 1 and in Table 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 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.

Regarding the analysis of the dataset without vulnerabilities, Tables 5 and 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: 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 6: Vulnerability detection in Juliet 32 bit dataset without vulnerabilities

6.2. CodeQL dataset

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

Table 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 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 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 10: Time taken to analyse CodeQL 32 bit dataset with vulnerabilities

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

Table 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 12: Vulnerability detection in CodeQL 32 bit dataset without vulnerabilities

The CodeQL dataset contained 19 programs vulnerable to heap overflows and 12 programs without any kind of vulnerabilities. Tables 7 and 9 show that HeapDUO was able to detect all the 19 vulnerabilities independently of the architecture being analysed with an average time of 4.51 seconds on *x86-64* and 4.50 seconds on *x86* as can be seen in Tables 8 and 10, respectively. Regarding the binaries without vulnerabilities HeapDUO did not generate any false positives. Similar to the Juliet test suit, AFL++ performed poorly due to the same reasons explained before.

6.3. Real World Vulnerabilities

GUEB showed its success by finding three vulnerabilities in open source software which were assigned an unique CVE, CVE-2015-5221 (Jasper-JPEG-200), CVE-2015-8871 (openjpeg) and CVE-2016-3177 (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 on *dmg2img*, 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.

7. 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 [15], and a symbolic engine AVD [18]. 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.

7.1. 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.

References

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

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

[3] Mitre, top 25 most dangerous software errors, 2021.

[4] Pwn2own results, 2021.

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

[6] G. Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), Aug. 2010.

[7] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[8] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[9] 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*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[10] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

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

[13] 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*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.

[14] 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*, SEC'18, page 99–116, USA, 2018. USENIX Association.

[15] J. Feist. *Finding the needle in the heap : combining binary analysis techniques to trigger use-after-free.* Theses, Université Grenoble Alpes, Mar. 2017.

[16] 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.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[18] N. Sabino. *Automatic vulnerability detection: Using compressed execution traces to guide symbolic execution.* Thesis, IST, Nov. 2019.

[19] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, Sept. 2005.

[20] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

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

[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*, ICSR'06, page 423–426, Berlin, Heidelberg, 2006. Springer-Verlag.

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