Hack the Heap: Heap Layout Manipulation made Easy

Jordy Gennissen
S3Lab, Information Security Group
Royal Holloway, University of London

Daniel O'Keeffe
S3Lab, Information Security Group
Royal Holloway, University of London

***Abstract*—Heap layout manipulation — the act of changing the heap layout to the benefit of the attacker — is a key challenge when exploiting heap vulnerabilities. Heap layout manipulation is a hard problem even for experienced exploit developers, due to the complex ways in which even simple operations affect the layout of the heap. Furthermore, different allocators can produce vastly different heap layouts for a given set of operations, and understanding these internal implementation differences is extremely time-consuming and error-prone. Existing work either focuses only on specific types of applications, requires access to source code, or produces complex and opaque solutions needing additional reverse engineering work to complete exploits.**

In this work, we propose Hack the Heap: an online puzzle game that provides direct solutions to the heap layout manipulation problem. The game comes with an extensive tutorial to guide players on how to play, but requires no Computer Science knowledge. Hack the Heap also provides a toolchain to generate heap vulnerability puzzles from unmodified real-world applications. We were able to create and solve puzzles from 2 out of 3 CVEs, both of which reflected a real-world solution to the heap layout manipulation problem. Hack the Heap makes heap layout manipulation easier and hopefully more fun, without sacrificing important information needed for practical exploit development.

I. Introduction

Computer applications are hardly ever perfect, and memory unsafe languages like C and C++ are still prevalent in today's infrastructure. Without memory safety, any violations, whether spatial or temporal, can introduce high-risk vulnerabilities. Not only can these vulnerabilities lead to arbitrary code execution, it is often extremely hard to assess their exploitability.

The best way of assessing the exploitability of a vulnerability is still to actually perform the exploit, creating ground truth. However, exploit writing is a specialised line of work that is extremely time-consuming. Although existing efforts have attempted to automate different aspects of exploitation [1], [2] or even a full pipeline of the process [3], [4], [5], [6], their use to fully generate exploits is limited. Furthermore, partly automating the exploitation process needs to provide the exploit writer with enough detail to finalise the exploit.

Within exploitation research and automation, most efforts have been tailored towards the stack [3], [4], [5], [7], [8]. Far less attention has gone to heap-based memory violations [9], [1], [10], which are generally more difficult to exploit due to the complex nature of the heap. In heap exploitation a key step for an attacker is to overwrite a useful data structure, e.g. a function pointer, data pointer or an `authenticated` flag (to bypass authentication), to enhance control over the application. Changing the heap layout so the vulnerability overwrites *this* data structure of choice is known as the Heap Layout Manipulation (HLM) problem.

The HLM problem poses several challenges. First, the memory layout for each heap manager is complex, and any heap interaction can completely change the placement of subsequent memory requests. For example, allocating a chunk of memory and immediately freeing it can affect the location of the next allocation. Second, implementation differences between different heap managers can result in different heap layouts for the same application. For example, the commonly used Linux heap manager PTMalloc2, Windows' heap manager LFH and TCMalloc (used by e.g. Google Chrome) work very differently internally. Thoroughly understanding the heap managers' behaviour is therefore vital for HLM, but also extremely time-consuming and error-prone. Third, the heap does not have a generic target to overwrite apart from the heap metadata, and metadata targets can be easily protected [11], [12]. Hence, we need to find memory chunks with data we want to overwrite, for example with the approach of Roney et al. [13].

In automatic solutions [1], [14], a fourth challenge arises. More often than not, the HLM problem is a single step in the bigger picture of creating the exploit. Thus, not only does the HLM problem need a solution, but also an *explainable* solution, so the exploit writer is not left with an opaque solution that needs to be reverse engineered before they can proceed.

In this work, we draw parallels between the HLM problem and traditional puzzles. From our perspective, the HLM problem shows many similarities with classic puzzles, where solving requires logic reasoning and thinking ahead. Puzzle games have been used to successfully crowd-source solutions to complex scientific problems in other domains [15], [16], [17]. Key to the success is to make the game *challenging and fun* while retaining the original goal of creating scientifically meaningful results. Besides, the visual nature of puzzle games often makes the solutions they produce easy to understand and interpret.

We propose "Hack the Heap": an online puzzle game that can represent both synthetic and real-world HLM problems in a visual puzzle format. Hack the Heap (HTH) can be played by anyone regardless of knowledge about the heap,

memory or any background in computing or computer science. HTH contains an extensive tutorial system to teach a player how heaps work interactively. HTH does not introduce any computer science terminology or other irrelevant complexity, so anyone can play. At the final level of HTH, players can solve puzzles generated from real-world programs using the HTH toolchain. HTH enables crowdsourcing the exploit writer's task by gathering solutions to the HLM problem over time. Once solved, the exploit writer can replay the solution(s) in the game interface to see how each solution works and choose the solution that best fits the conditions for exploit writing.

To summarise, we make the following contributions:

1) We convert the HLM problem into an online puzzle game called "Hack the Heap" (HTH). The game is paired with an extensive tutorial that non-experts can play to ultimately solve HLM problems without domain-knowledge. The HTH backend contains multiple heap managers, providing realistic behaviour for PTMalloc2, DLMalloc, TCMalloc and JEMalloc.
2) We develop a heap recording toolchain that can create HTH puzzles from real-world applications. The toolchain works without recompiling or altering the application and provides directly playable and accurate puzzles.
3) We create puzzles for different CVEs and show that solutions to these puzzles are indeed solutions to the real-world HLM problem.

II. Background

This section provides an overview of background information on the HLM problem and the use of games to benefit research.

A. The Heap Layout Manipulation Problem

The Heap Layout Manipulation problem, also known as Heap feng shui [18], [11] or heap massaging [19], is the attempt to set up the heap layout in a way that is beneficial to the exploit writer. The aim is to set up a heap layout such that the vulnerability will overwrite a memory chunk of interest. We call these chunks *targets*. Similarly, we have access to memory chunks that write outside the temporal or spatial boundaries. This can result from a temporal vulnerability (e.g. a use-after-free) or a spatial vulnerability (e.g. an overflow). We call the vulnerable memory chunk(s) *bugged*.

Example: Consider the code in Listing 1. This is a simple and poorly coded password manager with three operations. The `new` operation saves a new record containing a password to be saved; `alter_value` can change the saved password of a record; and `print` shows the saved password on the screen. The `new` function requires a password length as well as the password. This length is checked in the copying function on line 13.Yet, the size of this new buffer is set to twice the length of the string as seen in line 11. This creates the potential for an (unsigned) integer overflow where $2 \times \texttt{length} < \texttt{length}$ and hence the buffer would be too small for the submitted password. We refer to this allocation (on line 12) as a *bugged* allocation. Next, we need to find a useful value to overwrite with this overflow. No function/control pointers are available on the heap, but we do have one data pointer available: the `value` inside the record struct on line 4. If we overwrite this value, we can control both *where* to write and *what* to write by using the `alter_value` operation on the overwritten record afterwards. The allocation to the record struct creates this pointer on the heap (line 9), and thus we call this a *target* allocation. With the information above, the goal is to allocate a *bugged* allocation adjacent to a *target* allocation, on a lower address (so the overflow writes into the target allocation). *This is the heap layout manipulation problem.* A solution to our example is depicted in Figure 1 and is comprised of the following operations:

(i) `new` operation with size 24, twice. Shown in layout 1 of Figure 1.
(ii) `alter_value` operation on step (i) to size 64, as shown in layout 2 of Figure 1.
(iii) `new` operation with size 32780, that will overflow in an allocation of size 24 and writes into the second record created in step (i), see layout 3 in Figure 1. Any additional data written to this overflow will write into the record object, overwriting a data pointer and creating an arbitrary write.

Listing 1: Vulnerable example code

```
1  typedef struct {
2    size_t id;
3    uint16_t content_length;
4    char *value;
5  } record;
6
7  record *new(size_t id, char *password,
8        uint16_t length) {
9    record *rec = malloc(sizeof(record));
10   rec->id = id;
11   rec->content_length = 2*length;
12   rec->value = malloc(rec->content_length);
13   strlcpy(rec->value, password, length);
14   return rec;
15 }
16
17 void alter_value(record *rec, char *password,
18       uint16_t length) {
19   if (rec->content_length < length) {
20     free(rec->value);
21     rec->content_length = length;
22     rec->value = malloc(rec->content_length);
23   }
24   strlcpy(rec->value, password, length);
25 }
26
27 void print_value(record *rec) {
28   printf("Password for id %lu: '%s'",
29         rec->id, rec->value);
30 }
```

B. Existing Solutions and System Requirements

Heap layout manipulation has received attention in the research community only recently. SHRIKE [1] performs a random search for a solution, but is limited to interpreters and requires (re)compilation of the application. In follow-up work, Gollum [10] uses an evolutionary algorithm to enhance its success rate. SLAKE [20] performs heap layout manipulation on the kernel, using the kernel's heap management properties

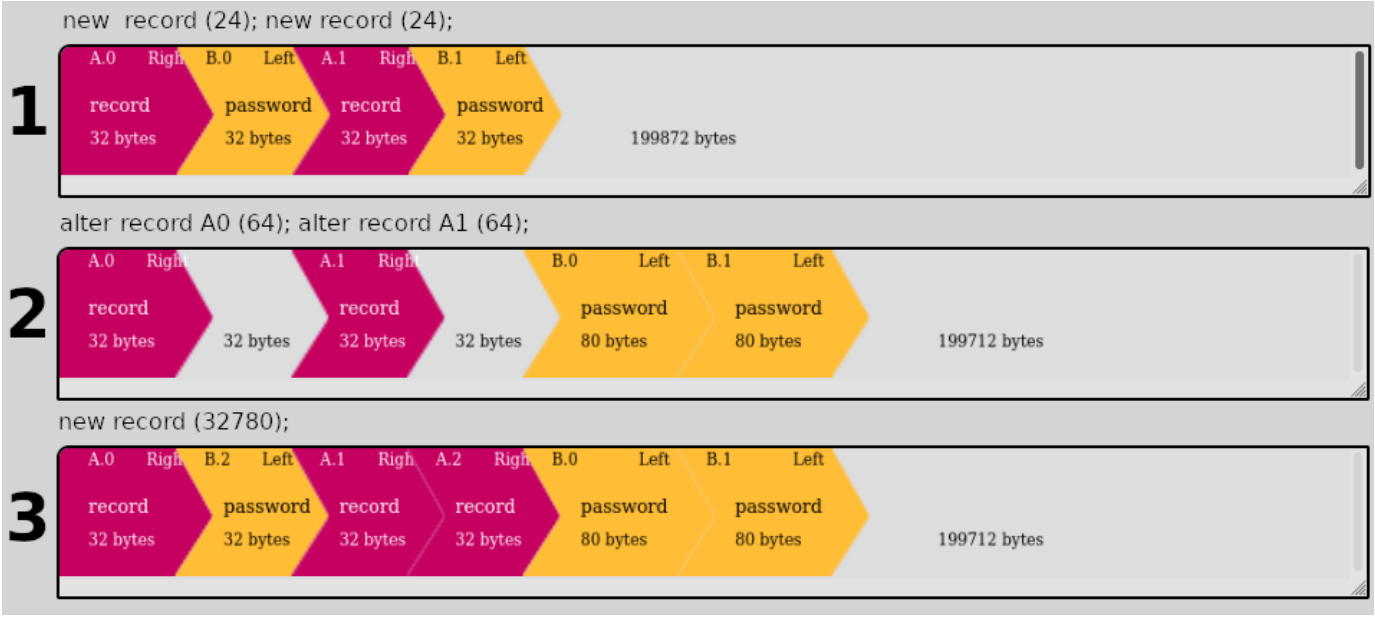


Fig. 1: A visualisation of an HLM solution as discussed in Section II-A. Between each step, the lines of code are shown how to go from one step to the next. In step 3, we are left with a "password" memory chunk (the left-most one) right next to a "record" memory chunk. An overflow from the password chunk would leak into the record chunk, overwriting a data pointer. The size of each puzzle piece is slightly bigger (e.g. 32 bytes) than the requested size (24 bytes) due to the added metadata from the heap manager.

to implement a custom search technique. Similar to SHRIKE and Gollum, it requires recompilation and is limited to a narrow set of applications.

Most recently, MAZE [14] proposes a custom "dig&fill" algorithm for the HLM problem. MAZE analyses operation-based applications using symbolic execution to automatically find the operations one can perform. It then probes the heap manager with similar (concrete) heap actions to find patterns in the operations: creating or filling "holes". This converts the problem into a list of constraints, to be solved with a constraint solver. Although MAZE does not appear to require access to source code[1], it still has a number of drawbacks. First, MAZE is limited to applications that can be reasonably symbolically executed, but symbolic execution is strongly affected by state explosion and at times, imprecision. Second, the dig&fill algorithm is closely tied to the inner workings of dlmalloc and ptmalloc (a threaded adaptation of dlmalloc). Finally, translation of operations into constraints can only occur if operations obey certain conditions with respect to the size of allocations. To ensure this MAZE employs different strategies (e.g. operation merging) that while convenient for the solver still limit the search space, potentially excluding solutions.

Orthogonal to these challenges, complete heap-based AEG solutions are still limited in their ability to tackle real-world programs without user interaction. The user requires a level of control to make sure that (1) the operations performed in the solution and their ordering make sense; (2) the resulting state of the application (and of the heap) is clear to the user; and (3) the state of the application after the HLM problem is solved is useful for continuing the exploit.

In short, we formulate the following requirements:

R1. The approach should work on a wide range of applications, regardless of application type, complexity or availability of source code.

R2. The approach should not be limited to a specific type (or types) of heap manager.

R3. The approach should not fundamentally discard parts of the solution space to simplify the problem without the ability to return to the original solution space if a solution is not found.

R4. The approach should provide insight into the HLM solution(s), providing all the necessary tools to the exploit writer to continue writing the exploit.

C. Crowd-sourcing Research

Crowd-sourcing has been used in several scientific domains to solve a range of research tasks. Researchers have sought to motivate their crowd through financial gains [21]; the idea of being part of a research breakthrough [22], [23]; or by entertaining them through a fun game [15], [16], [17], i.e. *gamification*. Where the former two are often repetitive and non-challenging tasks, gamification *relies* on the task at hand being challenging: something to e.g. replace sudoku with. These are branded as scientific discovery games, where the goal is to engage the general public in a series of puzzles that hold scientific significance.

[1]MAZE's implementation is not available at the time of writing.

Arguably the most successful scientific discovery game is FoldIt [17], [24], [25], [26], a protein-folding game that made discoveries in the aid of AIDS, Cancer and Alzheimers, to mention a few. This game is essentially an optimisation problem on the folding of proteins. The FoldIt team has done extensive research towards how to build such a game [27], [28] with over 850,000 players. They also identified key factors for building a successful game [27] that informed our own design.

III. The Game

In order to solve the HLM problem, we propose an online puzzle game for anyone to play. The puzzle board is a one-dimensional jigsaw puzzle representing the heap.

A. Design

The puzzle board as shown in Figure 2-1 will be filled with "puzzle pieces", equivalent to chunks allocated on the heap through a call to `malloc()`. In order to place the puzzle pieces on the puzzle area, a player performs *operations* by clicking buttons, as shown in Figure 2-2. These operations represent operations in real-world programs, such as invoking an API call. Operations consist of one or more calls to functions of the malloc-family (such as `malloc, calloc, realloc, free`, etc.).

The goal in the game is to place the bugged and target puzzle pieces next to each other in the game, representing a heap overflow (or underflow) into useful data. To add to the intuition of the game, we simply call these pieces the "left" piece and the "right" piece instead of bugged and target pieces. In Figure 2, the last piece (called "harmful") is a *right* puzzle piece, representing a *target* in the HLM problem. If this puzzle piece ends up on the right, adjacent from the piece with the *left* marker, the puzzle is solved.

B. Leveling up

To dive straight in the game can be hard for people without intrinsic knowledge about heap internals. As such, we built a level-based system with tutorials to aid people in playing the game and progressing. Our mascot explains the game with the most basic primitives to be able to play the game (see Appendix I). For example, only a basic overflow without additional requirements will be presented. Furthermore, puzzle piece placement is done with "first fit" logic, meaning that every next piece will be placed on the left-most fitting position. At the end of this tutorial, the player will be presented with a challenge. Solving the challenge will progress the player to *level 1* as shown on the left-hand side of Figure 2.

At this point, the player can play various level 1 puzzles. If the player wants a new challenge, they can press the "level up" button (as seen in Figure 2-5). This starts a new tutorial about a new concept (e.g. a new attack type), followed by a new challenge using this new concept. Solving the challenge will again progress the player towards the next level, where they can play puzzles that could include the new concept. We chose this design to let the player decide when it is time to progress further. In other words, players can decide their own pace. What type of puzzle is played is shown on the left of the players' screen (Figure 2-4). The concepts introduced at each level are shown in Table I.

Level	Concept introduced
Level 1	Basics (First Fit, Overflow)
Level 2	realloc() action added
Level 3	initialisation function added
Level 4	Next fit
Level 5	Best fit
Level 6	memalign() action added
Level 7	Overflow upon free'ing
Level 8	List fit (free lists)
Level 9	Overflow upon allocating
Level 10	Direct pipe to PTMalloc2 Service (fit)

TABLE I: Different levels in the game with concepts introduced at that level

Upon hitting level 10, a PTMalloc2 fit will become available that does not simulate its behaviour. Instead, the backend will perform every requested operation using PTMalloc2 and record its behaviour. The results are fed back into the puzzle game so we can ensure a correct representation of the problem. We currently support *PTMalloc2, DLMalloc, TCMalloc and JEMalloc*.

C. Visualisation Challenges

The puzzle game shows heap chunks as arrow-like puzzle pieces in different colours. Multiple issues arose in the design of this memory bar. First, we wanted the size of each puzzle piece to reflect its actual size (in bytes). The initial design scaled the different puzzle pieces to their actual size, but this proved unhelpful. Allocation sizes in real-world scenarios stretch over a wide range, creating excessively large chunks. Smaller chunks also became too small to fit all the information (name, amount of bytes, identifier, left/right marker). In some scenarios, puzzle pieces became too small to fit in the arrow-like structure, completely breaking the visualisation. This was solved by setting a minimum size of a puzzle piece (the "step" piece in Figure 2). In our current design, the size of a puzzle piece grows logarithmically, so a 1024 byte piece is 4 times as large as a 1 byte piece. With this solution, we cannot scale all the puzzle pieces anymore to fit in the memory bar. Thus, when the bar fills up completely, a second bar is created on the fly underneath. The player can either decide to scroll inside the memory bar or drag the bar down so multiple bars are shown at the same time. This is especially useful when analysing larger applications.

A second challenge is that operations can appear and disappear on the fly, even when performing seemingly unrelated operations. Consider a simple model with three operations: `x=malloc(128);` `realloc(x,512);` and `free(x)`. Upon performing the first operation, new instances of the realloc operation and the free operations will appear. Yet, when freeing the heap chunk, the realloc operation is not valid anymore. These conditional operations are shown on the left-hand side as puzzle details (Figure 2, label 3). Puzzle details include which operations create new types, what each operation requires to be executed, and what it may remove

Fig. 2: A screenshot of the heap puzzle game being played. The numbers correspond as follows: 1. is the playing board (or heap memory space if you like). 2. shows all operations that can be performed at the current stage of the game. 3. shows all requirements of operations before said operation can be used. 4. shows additional details about the puzzles. 5. is a link to the tutorial where the player can level up.

(potentially rendering other operations invalid). When searching for other operations to perform, players can always consult this to see what they can do to gain new operations and/or not to lose operations.

Finally, users experienced issues when operations caused changes to the heap they did not expect. In particular, it took time and frustration to find what new puzzle pieces had appeared where, and as a result they often got lost in the new scenario. Hence, we decided to highlight all new puzzle pieces after an operation is performed. This helped the players track their actions while playing the game, making the game significantly more intuitive.

IV. Generating puzzles

In order to play the game, we obviously need puzzles. The HTH game consists of both artificially created puzzles and real-world (recorded) puzzles. Both are represented in the same format when playing the game.

A. Artificial Puzzles

Most puzzles that represent a real-world problem are relatively complex and hard for new players to get into. For example, most real-world puzzles only make sense using a real-world heap manager, but a real-world manager only gets introduced at level 10 (see Table I). To lower the bar for players at the beginning of the game, we present easier puzzles. We generate puzzles for the different levels according to Table I. Puzzles are generated based on a range of options (e.g. what fitting technique will it use) and weights (e.g. whether to perform a `malloc` or a `memalign`).

Yet, this is unfortunately not enough to generate interesting puzzles. For example, if the "bugged" piece(s) and the "target" piece(s) only appear in the initialisation function, then there is no point in playing. Similarly, if the total size of the heap is too small (or too large to perform heap spraying), it can render the puzzle impossible or become increasingly frustrating respectively. On the other hand, if a target allocation directly follows a bugged allocation, its solution is trivial and does not challenge the player. All these corner cases are checked when creating puzzles to ensure the best quality puzzles and keep the player engaged.

B. Real-world Puzzles

Ultimately, the HTH game has to be used on real-world applications, as an HLM solution to a synthetic puzzle bears no significance. In order to play real-world HLM problems, we developed the HTH recording framework: a UNIX-based framework that can record heap usage of real-world unmodified applications using any heap manager to turn it into a puzzle. This is done in two steps, as shown on the left-hand side of Figure 3. First, we record all calls to the heap manager and save this as a raw trace.This trace is then given to the Refinery, which turns it into a playable puzzle.

Recording Heap Usage. In order to record all heap usage, we preload the HTH library. This library implements wrappers for all functions from the malloc family, to not alter the original program. Instead, it is preloaded with the `LD_PRELOAD` environment variable to make sure our wrapper-functions get called. The wrappers collect function arguments, return value, return address and the pid among others. This information is sent through a message queue to the HTH Recorder. The HTH library contains additional *optional* functions that can be used when linking the application to the library (i.e. if the source code is available). These functions can mark

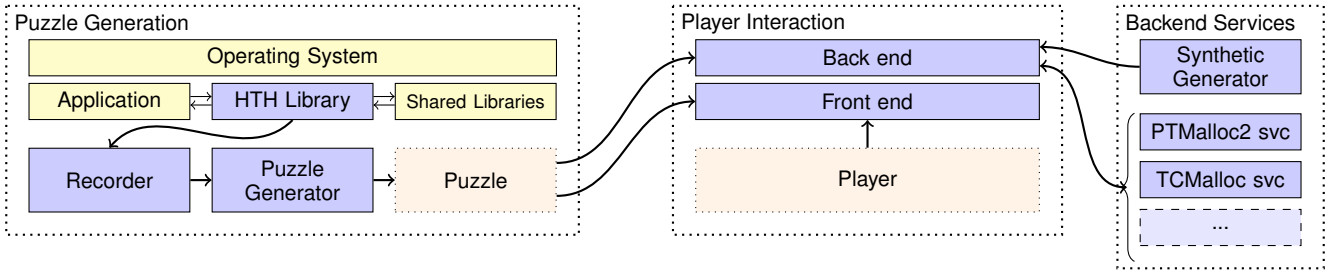


Fig. 3: The full infrastructure, split in three boxes. The left box describes the generation of real-world puzzles. The middle box describes the user interaction with the website, playing the HTH game. The right box describes the additional services running on the back end of the webserver. The blue colour denotes aspects we developed, whilst yellow shows systems untouched.

Listing 2: Recorder Output

```
1  SET target ./application
2  START
3  NEXT init
4  PROCESS_START
5    DYNAMIC /lib/x86_64-linux-gnu/libc.so.6
6          @ 0x7fe454ad1000 @ 0
7    MALLOC(128) @ 0x4011a9 = 0x99e2a0 @ 816848
8    MALLOC(1024) @ 0x7fe454b55e84 = 0x99e330 @ 816848
9    FORK 816848 -> 816849 @ 816848
10   MALLOC(4096) @ 0x7fe454b55e84 = 0x99e740 @ 816849
11 NEXT interact
12   REALLOC( 0x99e2a0 , 512) @ 0x401220 = 0x99f750
13         @ 816848
14   FREE( 0x99f750 ) @ 816848
15 END
```

particular mallocs with a custom name or type (bugged/target) statically, where this is marked in the recorder for later use. Alternatively, we offer this information manually, either by passing additional information to the Recorder interface or by manually altering the output later. On the first invocation of any of the overridden (or additional) functions, the library also attempts to locate the main function address. This is sent alongside the `PROCESS_START` keyword to mark a new execution, to calculate the ASLR offset to trace what line of code is being executed. For the same reason, any dynamic library used (that invokes our wrapper library) will be sent to the HTH Recorder. Together, this can be used later to leverage DWARF debug information (if available) to automatically determine the original variable name as used in the source code.

The information is captured and saved by the Recorder application. The Recorder can be controlled through a CLI, to run the target application, name and mark the start of various operations, interact with the target application through `stdin` and so on. This can be further automated with configuration files. Listing 2 shows an example Recorder output file.

The Refinery. We first need to split the recording based on the application's pids. For multiprocessed applications, any allocation before forking is present in both pids. After each fork, the heap layout will diverge depending on the pid we trace. As an example, Listing 2 line 9 shows a fork into a new pid: 81649. The subsequent call to `malloc(4096)` on line 10 is executed on this new pid. Similarly, the calls to `realloc` (line 12) and `free` (line 14) are not present in this child process. In the parent process trace, we do not want this `malloc(4096)` to appear. Splitting the file leaves us with one output file per pid, from where we can decide which to process.

These single-thread files can then be processed by the Refinery. In short, the Refinery parses the raw data into lists of heap actions per operation. In the parent process of Listing 2, this would be an `init` operation containing everything up to the `FORK` command, and an `interact` operation as denoted on line 11 (by the `NEXT` operation keyword) containing the `realloc` and the `free`. Afterwards, it traces the path each allocation makes (by tracing the return values) and labels each path with a unique label. The label tracks which `free`s and `realloc`s correspond with which earlier operations. This trace is colour-coded in Listing 2. Here we have three heap traces — yellow, pink and green — where the green one shows a longer trace of a `malloc`, a `realloc` and finally a `free`. In turn, it shows the dependencies, as every `free` requires the chunk to be allocated. After performing the `interact` operation, we cannot perform this again as the green trace has ended. Once we have the operations and labels, we generate a puzzle that can be played on the website.

Some real-world puzzles perform a very large amount of heap-related operations before the user interacts with the application, dubbed the initialisation operation. This is referred to by Heelan et al. [1] as noise: heap allocations that we cannot interact with throughout the course of heap manipulation. In some instances, it adds a burden to the person performing heap layout manipulation (i.e. adding difficulty to the puzzle). In other scenarios however, it adds "noise" to the visualisation and processing only, making it confusing to the player and adding significant loading time. To minimise this, the Refinery can remove some of these actions. In order to remain realistic and conservative, it only removes the middle allocation of three (non-bugged and non-target) allocations *if all three are not freed throughout the execution*. This simplification technique is optional and henceforth refer to this as the "simplified" puzzle.

V. Implementation

The game runs on a small Apache2 webserver, available to anyone at https://hacktheheap.io/. Besides the webserver, the game uses a database for saving (generated and submitted) puzzles and results. The puzzle game is implemented in TypeScript ES2015 and consists of approx. 5 KLoC. The layout is made with HTML5, CSS3 with Bootstrap and flexbox. The backend of the puzzle game is written in about 1 KLoC Python3 and C. The Heap recorder, library and refinery are written in C and Python3, consisting of approximately 2 KLoC.

VI. Evaluation

The goal of our evaluation is to show that we can represent and solve real-world HLM problems in the HTH puzzle. We evaluate the HTH infrastructure by generating puzzles from vulnerable applications. Afterwards, we attempt to solve the puzzles, solving the HLM problem for each heap vulnerability. The solution to the puzzles are applied to the original application to confirm it is a real solution to the HLM problem. All tests are performed on a 64-bit Ubuntu 18.04 VM with 4 cores and 8 GB of RAM, using the HacktheHeap.io website.

We first discuss a synthetic example application, taken from a Capture the Flag (CTF) challenge. Afterwards, we discuss the real-world application of Hack the Heap through 3 CVEs in NJS, the NGINX webserver's backend Javascript module.

Listing 3: Code snippet of the CTF challenge with an integer overflow on line 39–40.

```
30  typedef struct {
31    uint16_t id;
32    uint16_t content_len;
33    char *content;
34  } blob_s;
35  [..]
36  fread(&(new_obj->id), sizeof(uint16_t), 1, fp);
37  fread(&(new_obj->content_len), sizeof(uint16_t)
38      , 1, fp);
39  new_obj->content = malloc((uint16_t)
40      (2*new_obj->content_len));
41  fread(new_obj->content, 1, new_obj->content_len,
42      fp);
43  fclose(fp);
44  new_obj->content[new_obj->content_len] = '\0';
```

A. Case Study: Synthetic Example

In our first case study, we use a synthetic example from a non-public CTF challenge. The application provides us with a small interactive shell that can save strings on a given index. In this shell, we can perform various operations such as create a new object on a given id, edit the string, save the object to a file or load from a file. The developer arguably tried to avoid buffer overflows while loading an object by `malloc`ing exactly double the given content-length, creating an integer overflow. The trimmed code is shown in Listing 3, showing this vulnerability on lines 39–40.

When a content length is set over half the maximum size of a `uint16_t`, the calculation on line 40 will cause an unsigned integer overflow. Afterwards, lines 41–42 will cause a heap overflow on the `malloc`-ed memory. In puzzle game terminology, this means the puzzle piece created will be a "bugged" piece.

Next up is the question of what to target. Each object consists of an id, the content length and a pointer to the actual content — which is separately allocated. We aim to overwrite the content pointer of a separate object. If we can overwrite this pointer with a pointer value of our choice, we obtain an arbitrary write primitive. This can be used to overwrite the saved instruction pointer, a GOT table entry or to write a ROP chain onto the stack.

We record the various possible operations producing a recording similar to Listing 2. This is run through the refinery to create a puzzle to be played on the website. Playing the puzzle shows that we first need to perform a full object read before allocating the target, then delete the first read before reading the overflowing piece from a file. We can perform this in the original executable to gain the arbitrary write primitive. We finalise the exploit by overwriting the saved instruction pointer to point to a shellcode and gain a shell.

B. Case Study: NJS

NJS is a limited version of JavaScript, written to extend the functionality of the NginX webserver. In 2019, NJS was fuzzed by various fuzzers including Fluff, a JavaScript fuzzer by Samsung [29]. This lead to multiple CVEs, notably heap overflow CVEs CVE-2019-11839 [30]; CVE-2019-12206 [31]; and CVE-2019-13617 [32][2]. NJS is a particularly interesting case study as the amount of `free`s are limited within the user interactions. All CVEs have been patched.

In preparation for puzzle generation, we marked the allocation functions for objects that start with a pointer as target allocations. We use only allocations of objects that start with either a data pointer or a function pointer, so that an overwrite either creates an arbitrary write primitive or control-flow hijack respectively. Furthermore, the vulnerable version of NJS did not contain any functionality to recognise *what heap action (e.g. mallocs/frees) corresponded with what line* of JavaScript code. We wrote a small JavaScript module (of approx. 50LoC) to send additional messages to the recorder module (see Figure 3), marking the start of each individual JavaScript command. Using this, we can see what exact heap actions are performed with each line of JavaScript code. Finally, we marked the 'bugged' allocations for each of the CVEs (in separate runs), i.e. the allocations read/written to outside of its boundaries.

A variety of JavaScript commands were chosen as different operations to create the puzzle, each of which contain one or more calls to `malloc`/`memalign`/`free`, to present the player with a set of operations to alter the state of the heap. These always include one or more operations (i.e. lines of JavaScript code) that create bugged or target allocations. Every puzzle is available on the website to be played.

CVE-2019-11839 is a heap overflow on JavaScript arrays. When an array is created, space for an amount of (empty)

[2]CVE-2019-13617 was found through libfuzzer, the others by Fluff.

elements is allocated with an internal `size` variable. If an element is added to the array, it will check if there is still space available. If the allocated space is completely occupied, it will be reallocated into a bigger heap chunk. Alternatively, the first empty space will be used to store the new element. The `array.prototype.shift` operation removes an element from the beginning of the array, and is implemented to move the pointer of the start of the array one element forward. It failed however to update the internal `size` variable, as there is one element fewer that will fit inside the array at this point. A number of `shift`s on an array would cause its actual space to become far less than its internal `size` variable. Elements added at the end beyond its actual remaining size would erroneously not trigger a reallocation of the array, but instead cause a heap overflow until the `size` variable is exhausted.

Upon playing the simplified puzzle, we found a solution: First, we create a new array that we fill with elements until a resize takes place (10). Afterwards, we create a small regex object to fill up empty spots in the memory layout, before creating a JavaScript object. Internally, the JavaScript object starts with a data pointer that becomes directly adjacent to our array. A series of `shift`s will now misalign the size and free space as explained above, and subsequent `push`es to the array will overwrite the data pointer. The resulting heap layout is shown in Appendix III.

Performing the solution on the non-simplified puzzle shows that the same heap layout is achieved, confirming that the simplification did not alter the problem space nor its correctness, while purging 1119 allocations. However, context on how the results are achieved is important to an exploit writer, as this example clearly shows — the exploit writer can analyse the solution in the non-simplified puzzle, even when the simplified version is used to solve the puzzle. When writing a JavaScript file with the above solution, the desired heap layout is not achieved, as the JavaScript parser now creates a slightly different tree (with a different initialisation function, as our puzzle game is considered). A knowledgeable exploit writer does however see that this issue is easily solved by filling the gaps left with a few `malloc` calls. Rerunning the HTH Recorder on this new solution shows that the bugged heap chunk is directly followed by the target heap chunk.

CVE-2019-12206 arises when transforming strings to uppercase or lowercase variants. In particular, these functions do not distinguish between the byte size of a UTF-8 encoded string and its string length. For characters with a representation requiring more than one byte in UTF-8 (e.g. "è"), the string length becomes smaller than the byte size of the string. The new heap buffer created for said string (based on the string length) will be too small to fit all the bytes, leading to a heap overwrite. In contrast to the previous puzzle (from CVE-2019-11839), the overflow here occurs *directly after* allocating the 'bugged' memory chunk. In other words, the target chunk already needs to be in place when the bugged chunk is allocated, and cannot be allocated afterwards as it will overwrite the overflow data. This mode is the "Overflow upon allocating": level 9 in the HTH tutorial.

In this instance, we created an array and filled it until it grew once. Afterwards, we created a second, empty array to fill up the gap left from the resized first array. Next, we placed a target puzzle piece: this time we used a compiled regular expression object as target. The first fields of the compiled regular expression in NJS contains function pointers to malloc and free, to be used when performing a search with the regular expression. Overwriting these fields would give the attacker the power to call any executable section available.

To finalise the HTH solution, we grow the array again, creating a gap in front of the regular expression object. We finally transform a large UTF-8 string (with enough multi-byte characters) into lowercase, to overwrite the `malloc` function pointer into a value of choice. Here too, this solution is repeated in the non-simplified version and confirmed to produce the same results. In the same way, the solution in JavaScript code presents us with the desired heap layout for exploitation. A random search would have taken on average 84 days, see Appendix IV.

CVE-2019-13617 performs an overread when the lexer throws an error. Hence, it occurs during error handling upon parsing the JavaScript file (and hence before interpreting the actual code). Using our solution, this means that the bugged allocation occurs before any operation can be performed (and the application halts before that). Although the lexicographic differences can alter the heap layout when this vulnerability is triggered, our solution does not provide the necessary toolset to find an HLM solution. Within the puzzle, no operations are available (because it halts before any of the operations are executed) and the vulnerability is only available in the initial operation, before the player can perform any action at all.

VII. Discussion & Future Work

In Section II, we specified 4 requirements for our solution. **R1:** Our solution works on a wide range of applications, as long as different interactions can be represented as different HTH operations. The main bottleneck of HTH is the players' browser and internet speed when puzzles are large. **R2:** HTH supports four heap managers, and the HTH game and its recorder design is heap manager agnostic. This does mean that repeatability remains an issue in non-deterministic heap managers, as a solution may need to be repeated many times for the same result. **R3:** The Recorder performs only one simplification that does not change the problem space. Even in the case of distrust towards this or potential future simplifications, they are not fundamentally required to find a solution to the HLM problem. **R4:** Our solution visualizes the heap, providing insight into the heap layout. Different players can find different solutions, that show exactly how and why the solution works. The exploit writer can choose whichever solution works best to continue exploit development.

HTH could benefit from automatic solutions. For example, finding targets could be done by the solution as proposed by Roney et al. [13]. Marking the vulnerable section can be automated, for example using vulnerability patches as done by

Brumley et al. [3]. Finally, automatically extracting operations can be done through e.g. fuzzing [33], [29] or symbolic execution [14]. We leave the full implementation of this as future work.

User experience. We would like to further evaluate several aspects of the game's user experience. In particular, user understanding of the game; their enjoyment; their ethical understanding (i.e. making sure they understand that playing it is *not* unethical); and repeatability (i.e. the likelihood of players returning). While this has been done informally on multiple occasions throughout the development of HTH, no final user experience evaluation has been performed.

Recorder Extensions. Our evaluation shows part of our limitations as CVE-2019-13617 could not be solved with our work. This is because our methodology uses post-startup operations as interactions, which the CVE did not have. It could be possible to create HTH puzzles by having multiple runs of the application with slightly different initial inputs (e.g. parameters or environment). Comparing the different raw output files could allow to gather the heap-related effects and turn them into suitable operations, synthetically building the expected heap interaction sequence.

Game Extensions. The HTH game can be extended in a variety of ways. As mentioned, HTH could benefit from additional heap manager services, running e.g. ZEND or LFH. Further, HTH can be extended with temporal memory attacks (e.g. UaF or use-before-init attacks) to support other types of vulnerabilities, or to add a distance "success" metric from bugged to target piece.

VIII. Conclusion

In this work, we presented a solution to the heap layout manipulation problem, by posing the problem as a puzzle game. The game can be played online through a browser and is described in laymans terms with an extensive tutorial system. We also presented a heap recording infrastructure, where the heap interactions of real-world applications can be recorded. Applications with a spatial heap vulnerability can be recorded and turned into a puzzle, where the solution to the puzzle represents an accurate solution to the heap layout manipulation problem. We showed its real-world usage and limitations through different CVEs, with solutions that remained correct when applying them to the application. Not only does this aid exploit writers in their efforts to overwrite something useful with a heap overflow. It also serves as a severity assessment: a solved real-world puzzle poses a very beneficial scenario to exploit writers, as an overwrite or overread into memory of choice can go a long way in exploit writing.

Acknowledgements

Thanks to Manouk Locher for the website design concept, and various internal testers. This research has been sponsored by L3Harris TRL Technology and Royal Holloway, University of London as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

REFERENCES

[1] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *USENIX Security*. USENIX Sec, 2018.

[2] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2018.

[3] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *IEEE Symposium on Security and Privacy*. S&P, 2008.

[4] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," in *NDSS*, 2011.

[5] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy*. S&P, 2012.

[6] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *USENIX Security*. USENIX Sec, 2015.

[7] H. Shacham *et al.*, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." in *ACM conference on Computer and communications security*. CCS, 2007.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *ACM conference on Computer and communications security*. CCS, 2008.

[9] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Workshop on Programming Languages and Analysis for Security*. PLAS, 2017.

[10] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2019.

[11] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *ACM conference on Computer and communications security*. CCS, 2010.

[12] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *USENIX Security*. USENIX, 2018.

[13] J. Roney, T. Appel, P. Pinisetti, and J. Mickens, "Identifying valuable pointers in heap data," in *IEEE Security and Privacy Workshops (SPW)*. WOOT, 2021.

[14] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "{MAZE}: Towards automated heap feng shui," in *USENIX Security*. USENIX, 2021.

[15] L. Von Ahn and L. Dabbish, "Labeling images with a computer game," in *SIGCHI conference on Human factors in computing systems*, 2004.

[16] L. Von Ahn, R. Liu, and M. Blum, "Peekaboom: a game for locating objects in images," in *SIGCHI conference on Human Factors in computing systems*, 2006.

[17] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popović *et al.*, "Predicting protein structures with a multiplayer online game," *Nature*, 2010.

[18] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[19] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure," in *IEEE European Symposium on Security and Privacy*. EUROS&P, 2018.

[20] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2019.

[21] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna, "Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance," in *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2017.

[22] K. Land, A. Slosar, C. Lintott, D. Andreescu, S. Bamford, P. Murray, R. Nichol, M. J. Raddick, K. Schawinski, A. Szalay *et al.*, "Galaxy zoo: the large-scale spin statistics of spiral galaxies in the sloan digital sky survey," *Monthly Notices of the Royal Astronomical Society*, 2008.

[23] N. L. Eisner, O. Barragán, S. Aigrain *et al.*, "Planet Hunters TESS I: TOI 813, a subgiant hosting a transiting Saturn-sized planet on an 84-day orbit," *Monthly Notices of the Royal Astronomical Society*, 2020.

[24] F. Khatib, F. DiMaio, S. Cooper, M. Kazmierczyk, M. Gilski, S. Krzywda, H. Zabranska, I. Pichova, J. Thompson, Z. Popović *et al.*, "Crystal structure of a monomeric retroviral protease solved by protein folding game players," *Nature structural & molecular biology*, 2011.

[25] M. Gilski, M. Kazmierczyk, S. Krzywda, H. Zábranská, S. Cooper, Z. Popović, F. Khatib, F. DiMaio, J. Thompson, D. Baker *et al.*, "High-resolution structure of a retroviral protease folded as a monomer," *Acta Crystallographica Section D: Biological Crystallography*, 2011.

[26] F. Khatib, S. Cooper, M. D. Tyka, K. Xu, I. Makedon, Z. Popović, and D. Baker, "Algorithm discovery by protein folding game players," *National Academy of Sciences*, 2011.

[27] S. Cooper, A. Treuille, J. Barbero, A. Leaver-Fay, K. Tuite, F. Khatib, A. C. Snyder, M. Beenen, D. Salesin, D. Baker *et al.*, "The challenge of designing scientific discovery games," in *International Conference on the Foundations of Digital Games*, 2010.

[28] E. Andersen, E. O'Rourke, Y.-E. Liu, R. Snider, J. Lowdermilk, D. Truong, S. Cooper, and Z. Popovic, "The impact of tutorials on games of varying complexity," in *SIGCHI Conference on Human Factors in Computing Systems*, 2012.

[29] M. Dominiak and W. Rauner, "Efficient approach to fuzzing interpreters," *BlackHat Asia*, 2019.

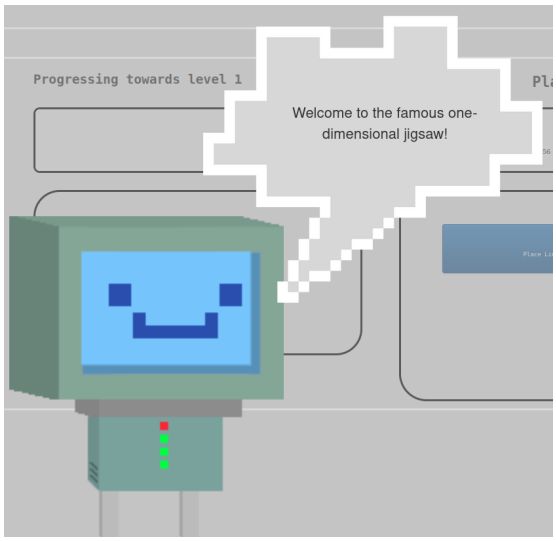
[30] "CVE-2019-11839." Available from MITRE, CVE-ID CVE-2019-11839., May 3 2019. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11839

[31] "CVE-2019-12206." Available from MITRE, CVE-ID CVE-2019-12206., May 3 2019. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12206

[32] "CVE-2019-13617." Available from MITRE, CVE-ID CVE-2019-13617., Jul. 3 2019. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13617

[33] M. Zalewski, "american fuzzy lop," http://lcamtuf.coredump.cx/afl/, accessed: 2020-07-21.

Appendix I. The HTH Mascot used in the tutorials



This is the HTH Mascot that guides the player in the tutorial. It also shows up to congratulate the player when successfully solving a puzzle, and when the user thinks that the puzzle is impossible to solve.

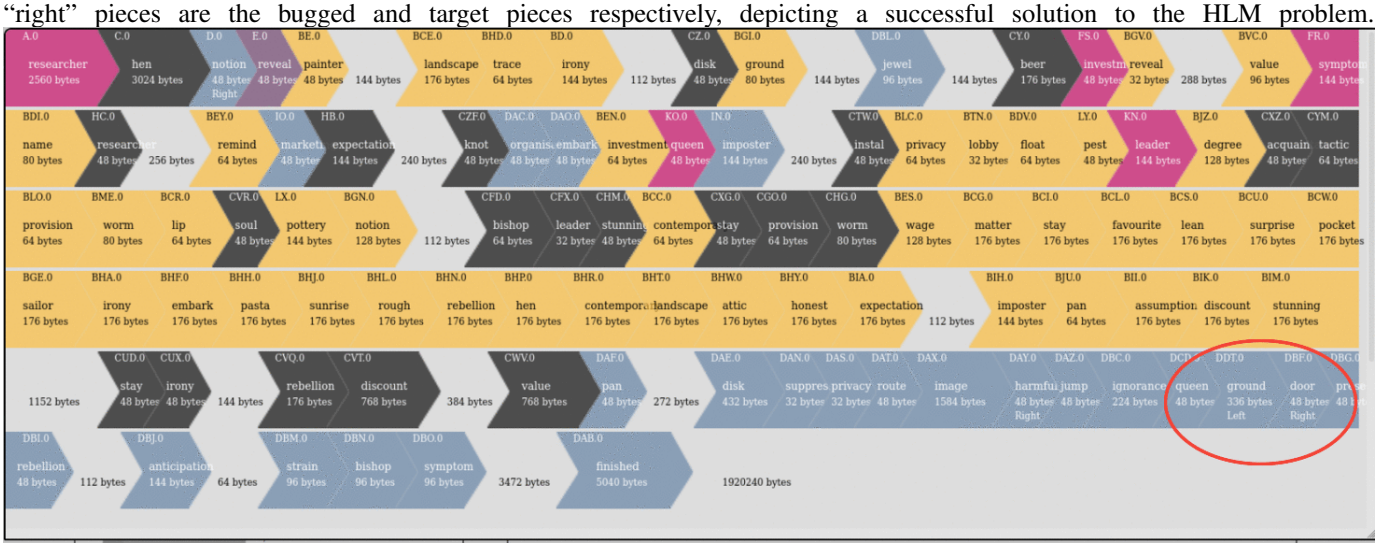
Appendix II. Context-free Grammar of the puzzle format

This is the context-free grammar used to represent a HTH puzzle in string format. It starts with a magic (`HPM2/`) followed by the heap manager F (e.g. first fit, ptmalloc, jemalloc). Afterwards, it describes the attack type A (e.g. OFA meaning overflow on allocation) and the size of the heap Z. This is followed by the individual operations O, with a name, colon(:) and a list of actions. Actions T are described by a number (e.g. 1 is a `malloc`, etc.) a tag to represent its trace followed by a name and argument(s) if applicable. Different operations are separated with a period(.), an additional period suggests that the operation is an initialisation operation.

```
S -> 'HPM2/'FAZO
F -> 'F'|'B'|'L'|'N'|'P'|'D'|'T'|'J'
A -> 'OVF'|'OFA'|'OFD'|..
Z -> [num]'T'
O -> O.O
   | [name]:T
   | .[name]:T
T -> TT
   | [0-4][TAG]P(R)
P -> &B
   | &T
   | λ
R -> [name]:[ARG]
   | [name]:
   | λ
[ARG] -> [ARG],[ARG]
       | [num]
[TAG]  -> [A-Z]+
[name] -> [a-zA-Z]+
[num]  -> [0-9]+
```

Appendix III. Heap layout solution from CVE-2019-11839

This figure shows the full layout of the heap of CVE-2019-18839 after performing the solution as discussed in Section VI-B. At the bottom right, two adjacent puzzle pieces are highlighted with a red ellipse. These "left" and "right" pieces are the bugged and target pieces respectively, depicting a successful solution to the HLM problem.



APPENDIX IV. ENUMERATION COMPARISON

When provided with a puzzle, it is difficult to compare the quality of the users playing against other techniques. We can however estimate the time it takes for either a depth-first search on the available operations or a random search. Estimating this precisely is not straightforward however due to dependencies: any operation that enables or disables another operation changes the options for the next choice of operation.

In the puzzle for CVE-2019-11839, we have a total of 16 base operations (that can be performed at any time). One operation (creating an array) enables a chain of operations (growing the array), each of which disable an operation. Creating two arrays would also create two additional operations. If we are able to keep growing our array (which is not in the current puzzle), the amount of operations can be described in the following formula.

$$\Sigma_{i=1}^{n}(x+i-1)^{n-i}(x+i)$$

Here, n denotes the amount of operations performed in the sequence, and x denotes the amount of non-changing operations: $x = 15$ in CVE-2019-11839 and $x = 16$ in CVE-2019-12206. If we check for a correct solution after every operation, we do not need to repeat smaller chains. Table II shows the time it takes to enumerate all options of a given length (after compensating for non-infinite growth). Here we assume that we can perform an operation followed by a check if the heap layout has been achieved 8 times per second per CPU, including all heap manager logic and ignoring the time it takes to perform the initial operation, which generally takes a significant amount of time. As the table shows, a full enumeration would take in the worst case approximately 6 CPU-days and 168 CPU-days for CVE-2019-11839 and CVE-2019-12206 respectively, and on average at least 3 and 84 CPU-days respectively.

# of operations performed max	1	2	3	4	5	6	7
CVE-2019-11839	2s	64s	24m	8h	**6d**	117d	2041d
CVE-2019-12206	2s	72s	29m	10h	9d	**168d**	3168d

TABLE II: Approximate CPU core time (in seconds; minutes; hours and days) to enumerate all possibilities for the given puzzles, when 8 full operations and heap layout checks can be performed per second. Highlighted are the set where our solution appear.