Memery: Analyzing Heap Memory for Fun and Profit

Nathan Contreras, Mridu Nanda, Noah Singer
Harvard CS 263

Abstract— **Memory reading vulnerabilities allow attackers to siphon sensitive data from a remote victim process, but time constraints and throughput limitations may necessitate a method to efficiently identify memory of interest. Toward this end, we design MEMERY, a black-box heap analysis algorithm that extracts information about the high-level program constructs of an executable with a memory read vulnerability. MEMERY follows chains of pointers to reliably detect both singly- and doubly-linked data structures (and any looping within them) and offers insights about the types of information stored in the data structures (including pointers to functions and character strings). Besides singly- and doubly-linked data structures, MEMERY's analysis can be used as a foundation for detecting many other chained data structures.**

I. INTRODUCTION

Analyzing the memory of a program is an area of interest to the security community because understanding the data structures it uses can yield information about its vulnerabilities. Function pointers, for example, are high-value targets to attackers given that their intentional corruption allows the attacker to gain control of the execution of a victim process. Moreover, reverse-engineering the data structures of a program might aid an attacker in quickly determining the location of sensitive information amongst vast amounts of memory in the victim process's address space.

Prior work in the reverse engineering memory access/allocation has focused on analyzing program binaries to understand the data structures in use. For instance, REWARDS [2] monitors the program's calls to well-known functions (with known signatures of parameter variable types) to determine the types of different variables used by the program. Furthermore, HOWARD [3] uses dynamic analysis of a program's execution and memory access patterns to determine the data structures used by a program. Unfortunately, such approaches are limited by the necessity of the source binary for the process to analyze.

Given the prevalence of high-value information stored in memory, other previous work have focused on creating exploits focused on siphoning that data from the victim. For instance, *Memory Cartography* attacks [1] work by building an ASLR-resistant relative map of variables in the memory of a browser renderer process. First, the attacker analyzes the data section of their own browser process which has loaded a website that the victim will visit in the future. The attacker classifies pointer-sized region of memory as pointer/non-pointer by checking each region to see if it points to memory allocated by the renderer. The attacker then creates a refined map by repeatedly rebooting the machine, building a new map, and removing pointers that do not appear in every map. This reduces false positives (i.e. randomly valued pointer sized regions that happened to point to allocated memory during an insignificant number of iterations of map-building). After a victim visits an attacker webpage, an attacker-created Javascript program chases pointers to a C++ vtable in the map created offline, revealing the absolute address of one item in the relative map and thus elucidating the absolute addresses of everything in the relative map. With this information, the attacker Javascript can find a desired location in the victim browser's memory, such as cookies or other sensitive data. *Memory Cartography*'s mapping approach inspires some of the core ideas behind MEMERY.

II. MEMERY

Our goal is the *black-box analysis* of heap memory; that is, we seek to extract information about high-level program constructs (in particular, linked data structures and function pointers) from heap memory *without analyzing the behavior of* *assembly instructions in the code segment*. We also avoid using standard debugging instrumentation, such as GDB, since there are established anti-reverse-engineering measures to detect debugging instrumentation and modify program behavior accordingly [4]. To this end, we introduce MEMERY, a tool for online heap analysis, with the goal of contributing meaningfully to the reverse engineering of arbitrary black-box programs. Section III describes the design of MEMERY.

MEMERY's threat model makes several assumptions:

- A *read vulnerability* that allows for access to arbitrary regions of memory, without causing segmentation faults.
- A way to leak a single absolute address of an object on the heap.

Additionally, MEMERY relies on various tunable parameters that govern, for instance, the size of the heap. Section IV details how we simulated these assumptions on toy victim programs; one major area of future work is relaxing all of these these assumptions (Section V). MEMERY's threat model allows for *non-execute (NX) bits* and *address space layout randomization (ASLR)*.

The primary focus of MEMERY is to identify structured information in (or referenced in) heap memory. In particular, MEMERY is able to discover and distinguish various structures (such as various subtypes of *linked lists* and *binary trees*). It also identifies values that appear to be character strings or function pointers.

One recurring theme of this paper will be that each of these identification problems is inherently poorly-defined, since the goal is to discover abstract and "semantic" patterns in memory usage with access only to raw memory values, and such patterns (such as "list" or "tree") can be mapped onto actual memory schemes in a massive number of ways. A consequence of this is that it is in general simple to construct programmatic constructs that exhibit desired behavior (such as "linked list") that are not flagged by MEMERY's heuristics, or to conversely build programs that do not exhibit a desired behavior but are indeed indicated by MEMERY. This will be further discussed in Section V.

III. DESIGN

A. *Definitions*

We review some definitions about data representations in memory:

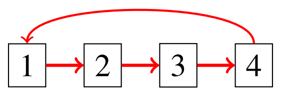
- **Linked data structure**: A localized pattern of memory usage (i.e. a C `struct`), where specific instances ("nodes") of the pattern are linked to each other by pointers. Linked data structures store relational information; for instance, they include *lists*, which store sequential information, and *trees*, which store hierarchical information.
- **Singly vs. multiply linked data structures**: A data structure is *singly linked* if each node contains one pointer to another node; it is *doubly linked* if each node contains two pointers to other nodes. A prototypical singly linked data structure is the linear *linked list*[1], where each node only stores a pointer to the next node; a prototypical doubly linked data structure is the *binary tree*, where each parent stores pointers to two of its children. Data structures can also be triply linked, etc.
- **Linear vs. circular data structures**: A *linear* data structure is one that has a defined "end" node, in which following pointers always terminates. In the *circular* data structure, every node is linked to another node.
- **Function pointer**: A pointer to an in-program function (used extensively in representing object-oriented constructs).

We will also define lower-level memory patterns. Let $M[i]$ denote the value of memory at an address i. Then we can identify the following objects:

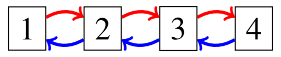
- **Chain**: A series of pointers $p_1, p_2, p_3, \ldots$ such that $M[p_1 + k] = p_2$, $M[p_2 + k] = p_3$, etc., where k is a small, positive value called the *chain offset*.
- **Loop**: A chain whose first and last elements are equal.

See Figure 1 for an illustration of pointers and chains for several common linked data structures. In these terms, primary goal of MEMERY is to bridge the gap between low-level constructs (chains and loops) and high-level constructs

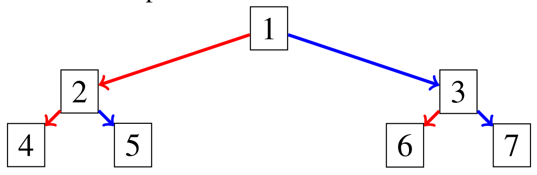
[1]Linked lists can also be doubly linked ("bidirectional").



(a) A singly-linked, circular list with four nodes. The red arrows are pointers from one node to another. There is a single chain of length four, and it is a loop.



(b) A doubly-linked, linear list with four nodes. The red arrows point from one node to the next node, and the blue arrows point from one node to the previous node. There are two chains of length four (formed by the red pointers and the blue pointers), and neither is a loop.



(c) A binary tree. The red arrows point from a node to its left child, and the blue arrows point from a node to its right child. There are four chains; two corresponding to each color (e.g. (1,2,4) and (3,6) for red).

Fig. 1: Three linked data structures, with descriptions of the corresponding chains.

(linked lists, trees, etc.) It can thus roughly be divided into two steps:

1) Discover chains.
2) Deduce information about the presence of structures.

Note, of course, that chains can form arbitrary graphs on the space of memory locations. Thus, we will consider the following two above problems "separately", in the sense that we will attempt to find chains in arbitrarily complex node networks, and then we will make heuristic guesses about what the chains might represent. These guesses will involve *invariants*, which are properties of collections of chains that "should be expressed" by a particular type of data structure (such as various forms of circularity).

B. Specification

We assume the victim process is running on a 64-bit machine, with addresses and data structure fields aligned on eight-byte boundaries, although MEMERY is not limited to this specific machine configuration.[2] We also assume that all linked data structures are allocated on the heap, and that the heap is a contiguous, accessible region of memory.

C. Singly-Linked Data Structure Algorithm

Address	Value	Heap Pointer?
0	100	
1	"Hello"	
2	3	✓
3	101	
4	"World"	
5	6	✓
6	102	
7	"!"	
8	9	✓

TABLE I: A toy memory representation of the beginning of a singly-linked linear list. Each node in the list stores a number and a string (e.g. (101, "World"), as well as a pointer to the start of the next node. When attempting to find a chain in this region, guessing an offset of $k = 1$ for the entry $p_1 = M[2] = 3$ at address 2 will fail, since $M[p_1 + k] =$ "World" is not a heap pointer. Conversely, guessing an offset of $k = 2$ will succeed, since $p_2 = M[p_1 + k] = 6$ and $p_3 = M[p_2 + k] = 9$ are indeed heap pointers.

The first step in MEMERY is to heuristically detect *singly-linked data structures*, which are interpreted as collections of overlapping chains (i.e. chains that contain common nodes). For instance, a *reverse binary tree* (i.e. every node contains a pointer to its parent) is a singly-linked data structure that consists of many individual chains.

Before singly-linked data structures can be detected, MEMERY must heuristically guess the starting and ending addresses of the heap. In our implementation, MEMERY has access to the absolute address v of some object on the heap, as well as an estimate H for the size of the heap. MEMERY heuristically treats all values in the range $[v - H, v + H]$ as *heap pointers*[3] and seeks to

[2]This assumption can be relaxed — MEMERY's algorithm would have to change to track potential data structures aligned at single-byte boundaries. Extra care would have to be taken to ensure that detected pointers wouldn't overlap.

[3]Assuming that H is large enough that the true heap falls into this range, there are no false negatives for detecting heap pointers; false positives are possible (either if H is too large, or if there is an integer that looks like a heap pointer but doesn't point to meaningful data).

construct chains out of such pointers. In particular, MEMERY iterates through all the potential pointers labeled above. On a given heap pointer p_1, it guesses possible values for a chain offset k (from 1 up to some fixed maximum offset). For each combination of p_1 and k:

1) Traverse the heap by setting $p_{i+1} := M[p_i + k]$ until one of the following conditions is satisfied:
   - A *loop* is detected: We encounter a pointer p_i that was previously encountered in this same heap traversal.
   - We arrive at a *pre-existing chain*: We encounter a pointer p_i that has been marked as belonging to an existing chain C_{old}.
   - The chain ends: We encounter a value p_i that is not within the range of potential heap pointers. (Typically, p_i will be null.)
2) If the traversal continues past some fixed minimum depth, we conclude that we have found a chain. We assign the traversed nodes to a chain; if we encountered a pre-existing chain C_{old}, we assign the nodes to that chain, or otherwise allocate a new chain and assign the nodes to that.

See Table I for a graphical representation of this process.

D. Classifying Node Contents

Once we determine the singly-linked data structures inside the heap, we want a way to classify the non-pointer contents of the data structure's nodes. In particular, if the pointer in a data structure occurs at offset k, we are interested in the information at positions $[0, k)$ in the data structure.[4]

MEMERY classifies the elements in the data structures as either `FUNC` (corresponding to function pointers), `STR` (corresponding to strings), or `INT` (the default type of each element). The following subsections address these classifications.

1) Detecting Function Pointers: Given our interest in scoping out function pointers in the victim's address space, we create a method for classifying pointers as function pointers with high certainty. To do so, we leverage the function calling-conventions of the victim's operating system/micro-architecture. Given a value p which we wish to classify as function-pointer/non-function-pointer, and our previously established ability to query for 8 bytes of data stored at any 8-byte-aligned address in the victim process's virtual memory, we proceed as follows:

1) Given the victim's suspected operating systems/micro-architecture, we define the function preamble as Pre. For example, on Ubuntu Linux 5.0.0-37 x86-64, the first three assembly operations in a function preamble are **PUSH MOV SUB ...**. In this case, Pre = **PUSH MOV SUB**. We also define Sz_{Pre} as the size of this function preamble pre in bytes.
2) We query for sz_{pre} bytes starting at p in the victim's machine. Assume we store these in an array $CandidatePtrBytes$.
3) We use a disassembler to output the sequence of assembly operations interpreted from $CandidatePtrBytes$, storing this sequence as $CandidatePre$.
4) If Pre is a prefix of $CandidatePre$, we know with high certainty that p points to a region of memory whose first bytes decode as a valid function preamble in the victim's machine, and p is a function pointer. If not, then the region is highly unlikely to correspond to valid function code, and p is not a function pointer.

2) Detecting String Pointers: We also wish to scope out pointers to character strings in the victim's address space, given that strings often contain sensitive information. Suppose the attacker is interested in certain types of strings that include characters in the alphabet Σ and are at least sz_{str} in length. For example, an attacker hoping to glean strings representing phone numbers would be interested in $\Sigma = \{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'\}$ with $sz_{str} = 10$ (given that phone numbers are usually 10 digits). Given a p which we wish to classify as a string pointer/not string pointer and the previously established method to query for 8 bytes of data stored at any 8-byte-aligned address on the victim's machine, we proceed as follows:

[4]We might also be interested in information that occurs after the pointer field; unfortunately, it is difficult to distinguish such information from other objects on the heap.

- We query for sz_{str} bytes (assuming each character is 1 byte) starting at p in the victim's machine. Assume we store these in an array $CandidateStrBytes$.
- We check if all sz_{str} characters are in $Chars$. If so, we know with high certainty that p points to a region of memory whose first bytes decode as a character string of interest on the victim's machine, and p is a string pointer. If not, then the region is highly unlikely to correspond to a valid string, and p is not a string pointer.

E. Multiply-Linked Data Structure Algorithm

Address	Value	Heap Pointer?
0	"Hello"	
1	3	✓
2	NULL	
3	"There"	
4	6	✓
5	0	✓
6	"World"	
7	9	✓
8	3	✓
9	"!"	
10	NULL	
11	6	✓

TABLE II: A toy memory representation of a four-element doubly-linked linear list. Each node in the list stores a string (e.g. "There") and pointers to the start of the next and previous nodes. Guessing an offset of $k = 1$ for the entry $p_1 = M[1] = 3$ at address 1 will succeed in finding the chain (1, 4, 7, 10). However, naïvely, an offset of $k = 1$ for the entry $p_1 = M[5] = 0$ will also succeed, since $M[0 + 1] = 1$ is in a pre-existing data structure; and so the algorithm will conclude that address 5 (as well as 8) is a pointer in the same singly-linked data structure. To fix this, we must ensure that singly-linked data structure nodes are non-overlapping.

The fundamental challenge when attempting to identify multiply-linked data structures, or *multi-structs*, is that they are difficult to distinguish from a singly-linked *out-of-order data* structure (i.e. where the in-memory order of the nodes differs from that which would be generated by a traversal). For instance, a doubly-linked linear list with n nodes might naïvely appear to be a singly-linked data structure with $2n - 1$ nodes, which overlap in pairs (see Table II). Although this ambiguity is unavoidable, we implement several additional heuristics in MEMERY's chain-finding algorithm to encourage it to correct identify multiply-linked structures. In particular, when searching from a given pointer p with purported offset k, we verify two additional conditions:

1) If we identified p as a memeber of a pre-existing chain C_{old}, we check that no pointer in $[p - k, p)$ is already assigned to C_{old}.
2) We check that no pointer in $[p - k, p)$ is already assigned to a chain with offset k.

If either of these conditions occurs, we conclude that we have selected the wrong offset k and so do not assign p to that chain. This means that, assuming that we have already seen a sufficiently large part of the data structure generated by pointers at one offset (e.g. we have seen several nodes linked by next pointers in a doubly-linked list), we are able to avoid assigning pointers at another offset to the same data structure (e.g. we can avoid previous pointers being identified as part of the same singly-linked data structure as next pointers in an in-order, doubly-linked list). See Section V for a discussion of where this technique can fall short.

Once we assemble all the chains, we must begin to identify multistructs. We simply consider the chains as the vertices of a graph, and there is an edge between two chains iff they intersect at at least one element; then the multistructs correspond to the connected components of this graph.

F. Invariants

Finally, MEMERY must attempt to identify the "high-level type" of data structure for each multistruct. We begin computing a number of properties of a given multistruct, such as the number of distinct nodes and the number of distinct chain offsets. (The latter quantity tells us whether the multistruct is "singly-linked", "doubly-linked", etc.) Then we calculate a number of *invariants*, which are high-level properties of structures that can be used to distinguish between classes of data structures. In particular, we employ:

1) *Overall strong connectedness*: Using all the chains in the multistruct together, is there a path from every node to every other node?

2) *Offset-wise strong connectedness*: When restricting to chains of each offset k, is there a path from every node to every other node?

See Table III for an example of how these invariants are useful in distinguishing doubly-linked data structures. All of these strong connectedness calculations are performed using a simplified variant of *Kosaraju's algorithm* for finding strongly connected components; in particular, we need only pick a given node n and check, via depth-first search, that every node is accessible from n via the original chains, and that every node is accesible from n when the direction of every edge in the chains are reversed.

	Overall ✓	**Overall ✗**
Offset-wise? ✓	Circular list	N/A
Offset-wise? ✗	Linear list	Binary tree

TABLE III: Various types of doubly-linked data structures, classified by overall strong connectedness and offset-wise strong connectedness.

IV. Implementation

We have developed an open-source implementation of MEMERY, available on GitHub at `tothepowerofn/memery`. In this section, we discuss relevant details of our implementation, with particular focus on how we simulated the vulnerabilities we assumed in Section II.

We developed a variety of test-cases representing linked lists that were singly and doubly linked, and linear and circular; for robustness, we tested where nodes were allocated in random or linear orders, and with random gaps between nodes. We also tested trees, in particular "parent trees" where each child only stores a parent pointer, as well as standard binary trees where each parent stores pointers to each of its two children. All tests were successful (up to the limitations in double-link detection described in V).

The interaction of the actual MEMERY engine with the victim program was simulated via a network interaction; the use of a simulated exploit-via-socket mirrors the real-world possibility of exploiting a webserver. The victim program listens for incoming connections. Upon a connection with MEMERY, the following steps take place:

1) The victim calls `malloc(1)` and transmits the result to MEMERY, simulating the leakage of a heap address via heap feng shui or a similar technique.
2) MEMERY queries the victim for the value stored at a certain address p. The victim checks that this is indeed an accessible address[5], and if so, it dereferences it and returns the result to the server.

After using this vulnerability to leak the entire contents of the "estimated heap" region, MEMERY proceeds with the algorithms described above to identify data structures, and then generates a user-friendly report on its findings.

V. Limitations and Future Work

One specific limitation of the above heuristics for detecting multiply-linked data structures has to do with the detection of out-of-order multiply-linked data structures, which consist of nodes whose memory addresses are not strictly increasing or strictly decreasing when traversed in the linked order. For instance, if the nodes in Table II had occurred in the order ("World", "!", "There", "Hello"), the next pointer for the "World" node with offset 1 would fail to generate a chain of sufficient depth (since following the pointer and adding 1 would lead to a NULL value in "!"); therefore, the first chain would be created by following the previous pointer for "World" with offset 1 into "There" and then following the next pointers for "There" and "World" to the null value in "!". This problem represents an inherent ambiguity in the detection of doubly-linked data structures (since the data could in fact form an out-of-order singly-linked data structure); solving this problem would necessitate implementing a sophisticated branching and backtracking analysis, which might significantly impair MEMERY's memory usage and runtime.

It is also not very difficult to design programs to intentionally confuse or evade the watchful eye of MEMERY. For instance, to implement a singly

[5]We employ the ingenious method described at this StackOverflow post for checking whether an address is readable in Linux: We setup a dummy memory file and then attempt to write a single byte from address p into the file. If the address is readable, the write will succeed, while if it is unreadable, the call to `write` will fail with error condition `EFAULT`.

linked list, a clever program could store, in each node, several slots for pointers. All these slots would be null, except for a single slot, the index of which would be randomly chosen and stored per-node, storing the pointer. Using this index to retrieve the pointer and advance to the next node, the program would be able to implement the *semantic properties* of a linked list (i.e. this is a valid implementation of a "linked list API"); however, the probability of chains of detectable length existing is very low, and so MEMERY would not be able to identify the data structure. Similarly, an adversarial program could populate large memory regions with **PUSH MOV SUB**; MEMERY is incapable of distinguishing such regions from true function pointers that would be useful for reverse engineering or further exploitation. While such issues are concerning from a robustness standpoint, it is a fundamental limitation that it is impossible for programs to truly reason about semantic properties of arbitrary code, and we believe our contribution is still meaningful and useful even in the non-adversarial case.

One potentially fruitful area for future research is in relaxing the assumption that we can perform arbitrary reads without causing segmentation faults. Right now, MEMERY proceeds by attempting to leak the entire estimated heap region; however, if the heap estimate is too liberal (and it is in the above design), an immediate segmentation fault will ensure. (Furthermore, a segmentation fault causes addresses to be re-randomized via ASLR, which would force MEMERY to have to figure out how to leverage cross-execution, relative-addressed-based learnings.) A more conservative strategy would progressively expand the leaked region while accessing pointers that look like they could potentially be heap pointers; when it comes to checking string and function pointers, perhaps using probabilistic reasoning about regions that are most often pointed to and would therefore be the most valuable to access would be helpful.

Another area is to try and recognize more sophisticated layouts of individual nodes within a structure. Currently, MEMERY is mostly suited towards recognizing linked data structures built from C `structs`; it cannot recognize C++ objects, including linked structures built using the C++ *Standard Template Library (STL)*. We believe that it would not be difficult to adapt our algorithm to, for instance, specifically search for and parse virtual table pointers (used to back C++ objects).

VI. CONCLUSIONS

In conclusion, MEMERY provides useful black-box analysis of heap memory and the data structures contained within it using only a memory read vulnerability in an NX- and ASLR-protected executable. It implements heuristics for the detection of singly-linked and multiply-linked data structures, and can identify function pointers and other types of useful data. We have successfully run MEMERY in a simulated, socket-based environment and demonstrated its ability to detect and distinguish linked data structures. We recognize the current limitations of MEMERY and hope to improve its robustness and expand its repertoire of data structure recognition in the future.

REFERENCES

[1] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, "Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses," *2017 IEEE European Symposium on Security and Privacy (EuroS&P),* 2017.

[2] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," *InProceedings of the 11th Annual Information Security Symposium,* 2010.

[3] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a Dynamic Excavator for Reverse Engineering Data Structures," *NDSS,* 2011.

[4] M. N. Gagnon, S. Taylor and A. K. Ghosh, "Software Protection through Anti-Debugging," in *IEEE Security & Privacy,* vol. 5, no. 3, pp. 82-84, May-June 2007.