Identifying Valuable Pointers in Heap Data

James Roney
Harvard University
Cambridge, MA

Troy Appel
Harvard University
Cambridge, MA

Prateek Pinisetti
Harvard University
Cambridge, MA

James Mickens
Harvard University
Cambridge, MA

{jamesroney, troyappel, prateek_pinisetti}@college.harvard.edu, mickens@g.harvard.edu

***Abstract*—Historically, attackers have sought to manipulate programs through the corruption of return addresses, function pointers, and other control flow data. However, as protections like ASLR, stack canaries, and no-execute bits have made such attacks more difficult, data-oriented exploits have received increasing attention. Such exploits try to subvert a program by reading or writing non-control data, without introducing any foreign code or violating the program's legitimate control flow graph. Recently, a data-oriented exploitation technique called memory cartography was introduced, in which an attacker navigates between allocated memory regions using a precompiled map to disclose sensitive program data. The efficacy of memory cartography is dependent on inter-region pointers being located at constant offsets within memory regions; thus, cartographic attacks are difficult to launch against memory regions like heaps and stacks that have nondeterministic layouts. In this paper, we lower the barrier to successful attacks against nondeterministic memory, demonstrating that pointers between regions of memory often possess unique "signatures" that allow attackers to identify them with high accuracy. These signatures are accurate even when the pointers reside in non-deterministic memory areas. In many real-world programs, this allows an attacker that is capable of reading bytes from a single heap to access all of process memory. Our findings underscore the importance of memory isolation via separate address spaces.**

I. Introduction

Memory bugs are common attack vectors. Over 70% of the CVEs patched by Microsoft each year are related to memory safety [1], and memory vulnerabilities like HeartBleed have caused international panic [2]. Such vulnerabilities often serve as starting points for attacks that subvert a program's control flow. For example, in a return-oriented programming attack, a malicious actor repeatedly overwrites return addresses to chain together short sequences of instructions and perform arbitrary computations [3]. Despite the continued prevalence of memory vulnerabilities [1], exploiting them to hijack program control flow has become significantly more difficult. Protections like Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) significantly hinder an attacker's ability to inject arbitrary code or locate executable gadgets; defenses like stack canaries and shadow stacks can detect violations of control flow integrity [4].

The increased difficulty of subverting a program's control flow has spawned increased interest in data-oriented attacks [5]. Data-oriented exploits refrain from modifying control data like return addresses and function pointers, thereby respecting the program's control flow graph. Instead, these attacks manipulate non-control data, like program variables and function arguments, to indirectly influence program execution.

The most basic data-oriented attacks directly manipulate variables that help decide program behavior. For example, Chen et al. [6] demonstrated an attack against SSH in which a memory vulnerability was used to overwrite a boolean flag containing the user's authentication status. This caused the server to treat the user as if the user had successfully authenticated, despite the user never having presenting valid credentials. More sophisticated attacks chain together gadgets inside of a program's legitimate control flow graph to perform complex manipulations of data [7]. Each gadget is a short sequence of basic blocks that performs a simple computation. These gadgets are invoked in sequence via the repeated manipulation of non-control data (often within a loop). This allows the attacker to perform highly flexible computations without violating the program's control flow integrity. Such multi-stage attacks are collectively referred to as "data-oriented programming" (DOP).

Data-oriented attacks have historically required extensive knowledge of the victim program and manual inspection of victim code [6]. Researchers have tried to automate the identification of gadget chains that facilitate DOP attacks, but initial efforts have been highly dependent on the exact nature of the memory vulnerability being exploited [8] [9]. Rogowski et al. [5] introduced *memory cartography*, a data-oriented attack that is largely agnostic to the specific vulnerability being exploited; the technique also requires minimal semantic knowledge of the victim program.

In a memory cartography attack, an attacker runs the victim program offline, examining the allocated memory regions using a tool like GDB. We use the term "memory region" to refer to a virtually contiguous set of pages within the victim process. For example, in Linux, such a region is called a virtual memory area (VMA). VMAs include code and data segments from the victim binary, as well as memory regions that are dynamically created through calls to `sbrk` and `mmap`. In Linux, the virtual memory ranges corresponding to each VMA can be obtained from the pseudo-file `/proc/<pid>/maps`.

After determining the bounds of each VMA, the attacker scans each region, checking whether each aligned, pointer-size value references a valid, external memory region. Such pointers link a specific offset in a source memory region to another offset in a destination region. In aggregate, these edges form a network that connects all regions of program memory. Because the attacker lacks source code or debugging symbols, the attacker cannot determine which edges are "false" edges, i.e., pointer-sized regions of memory that are not pointers, but whose binary values (when interpreted as pointers) reference valid memory regions. To eliminate such false edges, the attacker merely needs to load the program multiple times, building a new map for each load. Since (1) ASLR preserves the *relative* offsets of each true pointer within a memory region, and (2) a false pointer is unlikely to point to a valid memory region across multiple ASLR loads of a program, the attacker can eliminate false pointers after a handful of program loads. The result is a reliable map of links between memory regions. Armed with this map, a memory read vulnerability can be used to navigate from a source memory region to the rest of program memory without triggering a segmentation fault, even in the face of ASLR. This ability gives the attacker freedom to search for sensitive data anywhere in the address space.

The efficacy of memory cartography is dependent on inter-region pointers being located at consistent offsets within their host memory regions. While this is a valid assumption for memory regions containing static data, the offsets of program variables can be highly nondeterministic in regions like heaps and stacks. For this reason, Rogowski et al. primarily built their memory map using links between data sections, and only recorded incoming edges for heap regions. To use such a map, an attacker traverses multiple data sections before arriving at a pointer to a heap containing sensitive data. While this pointer may lead to an inconsistent offset within the heap, the attacker can simply search the surrounding region of memory for sensitive information.

The inability of memory cartography to build edges leaving a heap region can be a significant limitation. If an attacker cannot traverse heap regions to reach memory destinations, this may reduce the connectivity of the memory map. Perhaps more importantly, memory read vulnerabilities often originate from objects on the heap; thus, a cartographic attacker would need to initiate the attack by finding a heap-to-data-section pointer in the heap. When exploiting a web browser, Rogowski et al. overcame this obstacle by having their malicious web page allocate a large number of recognizable JavaScript objects on the heap. Rogowski et al. could find one of these objects by scanning the heap, and then follow a vtable pointer from that object to a static data region.

This approach is viable in the case of web browsers, where the attacker has fine-grained control of the objects that are allocated on the heap. However, many applications do not afford an attacker such control. We focus on these applications in this paper. In particular, we demonstrate how a cartographic attacker can identify pointers to data sections within non-deterministic heap data.

A. *Threat model*

The attacker's goal is to navigate from a program heap to a data section without causing a segmentation fault. We assume the following threat model:

- The attacker can read memory addresses at arbitrary offsets from a base address, which resides in the heap.
- The attacker can learn the base address, allowing them to read memory at absolute addresses by specifying an appropriate offset.
- The attacker can run the victim binary locally.
- If the attacker attempts to access an unmapped memory address (or other memory that is not user-accessible) a fault is triggered, and the attack fails.
- ASLR, DEP, stack canaries, and other control flow guards are enabled.
- The attacker does not have access to the source code of the victim program.
- The attacker's local environment is sufficiently similar to the victim machine that inter-region pointers are located at the same offsets in libraries on the local machine and the victim machine.

All but the last assumption are eminently reasonable, since many target programs have readily available binaries, have control flow guards enabled, and have historically suffered from memory vulnerabilities that allowed heap reads. The last assumption about environmental stability is crucial for the successful execution of memory cartography attacks, since inter-region pointers within static data sections must reside at the exact same offset during the offline and online phases of the attack [5]. Providing a full analysis of environmental stability with respect to OS type, CPU model, and so on is outside the scope of our paper. However, we note that our assumptions of environmental stability are no stronger than the stability assumptions made by Rogowski et al. in their original exposition of memory cartography. It should also be noted that our threat model makes no assumptions about the heap allocator being used by the victim program. In particular, our method works on programs where all heap objects are allocated in the same VMA, or via `mmap`-based allocators which create new VMAs to store dynamically allocated objects.

B. *Contributions and Paper Structure*

We demonstrate that, under our threat model, an attacker can reliably identify pointers to specific offsets within specific data regions of memory, and follow those pointers to launch a powerful memory cartography attack. In particular, we show that a simple signature-matching algorithm which learns the distribution of bytes surrounding pointers to the data section is sufficient to reliably identify those pointers in heap memory.

Part II gives an overview of relevant prior work on memory analysis. Part III describes our novel pointer-identification method. Part IV details the accuracy of this method on real-world programs, and Part V discusses the implications of our results.

II. Related Work

To the best of our knowledge, no prior work has attempted to automatically identify pointers to particular destinations from dynamically-allocated regions of memory. However, a substantial body of prior research has focused on identifying specific types of data structures in memory snapshots.

Finding in-memory kernel objects (and verifying their integrity) has been particularly well studied due to the importance of detecting rootkits that tamper with kernel data. Early approaches like Gibraltar [10] and KOP [11] located kernel objects by following pointers from statically-allocated global data structures, inferring the type of each object using static typing information from the kernel source code. After locating and typing data structures, these approaches assessed the integrity of each structure using invariants that were either hand-crafted or learned from observing objects in a benign kernel.

Other methods have been developed to discover kernel objects that cannot be reached from global data structures, or whose types cannot be robustly inferred from the kernel source code (as may be the case when a rootkit is attempting to hide a hijacked data structure). Dolan-Gavitt et al. [12] learn signatures associated with various kernel data types and match those signatures with heap memory to discover objects. To determine such a signature, the authors fuzzed the fields of the relevant kernel data structure and identified sensitive fields which caused the kernel to crash when fuzzed. They then built a signature by recording invariant properties of those sensitive fields during normal kernel operation. The use of sensitive fields makes the signature more robust, since any rootkit that manipulates an object's data to avoid matching the signature will likely trigger a system crash. Subsequent work has attempted to discover even more robust signatures for kernel objects by analyzing the pointer relationships between kernel objects of various types, and then scanning memory for objects which satisfy those relationships [13][14].

Other studies have looked for different kinds of structures in memory. For example, Graziano et al. [15] used a signature-matching algorithm to identify Virtual Machine Control Structure (VMCS) pages in a memory image, allowing them to locate and analyze the physical address spaces of all VMs running on the machine. MemPick [16] instruments a binary to record all buffers allocated by `malloc`, and analyzes the evolution of pointer relationships between these buffers to determine the types of data structures being used by the program. Laika [17] uses a Bayesian unsupervised learning algorithm that clusters segments of a memory image into a number of learned classes, each of which corresponds to a different data type. This algorithm can recover ground-truth pointer relationships between data types in real-world programs. It can also detect malware-infected processes with impressive accuracy by comparing a process's data structures to those of known malware samples.

Our efforts to identify important pointers in heap memory differ from the aforementioned methods in a number of ways. The signature-matching algorithms for kernel objects have impressive accuracy, but they require either source code or debugging symbols to infer the types of at least some kernel objects. Methods that do not require external type information, like MemPick, still require sophisticated binary instrumentation to detect all memory allocations. This contrasts with our method, which learns memory signatures from fully unannotated memory snapshots. In this respect, our work is most similar to Laika. However, when compared with the near-perfect accuracy of supervised methods for detecting kernel objects, the fully unsupervised nature of Laika's learning process produces lower true positive and true negative rates when determining object types. Our method achieves much higher accuracy, which is crucial if an attacker wants to confidently follow discovered heap pointers without fear of triggering a segmentation fault. Finally, rather than finding specific data structures in memory, we investigate the related problem of identifying pointers to specific destinations, and examine the repercussions of such identification methods in an offensive, rather than forensic, setting.

Our threat model is the most similar to that of Morton et al. [18]. Those authors demonstrated that an attacker capable of reading data from the heap of a web server could discover a pointer to an important table of server configuration data; the attacker could then leverage an arbitrary write vulnerability to reconfigure and hijack the server. The authors identified the pointer of interest by matching a signature that was built through manual analysis of the server's source code. Our work can be viewed as an attempt to automate these kinds of attacks, *even when no source information is available.*

III. Methods

We hypothesized that, despite the unpredictability of heap allocations, pointers to particular destinations would be identifiable by the data surrounding the pointer variables. We believed this to be true because two major sources of pointers from heap regions to data regions are function pointers and virtual table pointers, both of which are frequently embedded within objects or structs. Such objects and structs are likely to surround pointers with recognizable data types and values. For instance, data structures may contain string constants, magic numbers, null bytes, or other features that signal the presence of an adjacent pointer to a specific destination. Such regularities can be exploited by an attacker to find high-value pointers on the heap, even in the presence of ASLR and nondeterministic allocations.

Based on this hypothesis, we developed an algorithm to scan heap memory and identify pointers to specific data section offsets. At a high level, the algorithm executes the binary offline to learn the memory signatures that surround pointers of interest. Then, in the online phase, the attacker matches bytes from the heap to precompiled signatures, identifying valuable pointers. This process is illustrated in Figure 1, and a detailed description of the algorithm is as follows:

- **Step 1: Determine Allocations.** The attacker begins by starting the victim binary, obtaining the PID of the targeted process, and examining the information in the pseudo-file `/proc/<pid>/maps`. This file contains the ranges of the process's virtual address space that have been allocated, as well as names corresponding to those regions. The attacker stores this information in a list of tuples of the form

  `(region_start,region_end,region_name)`

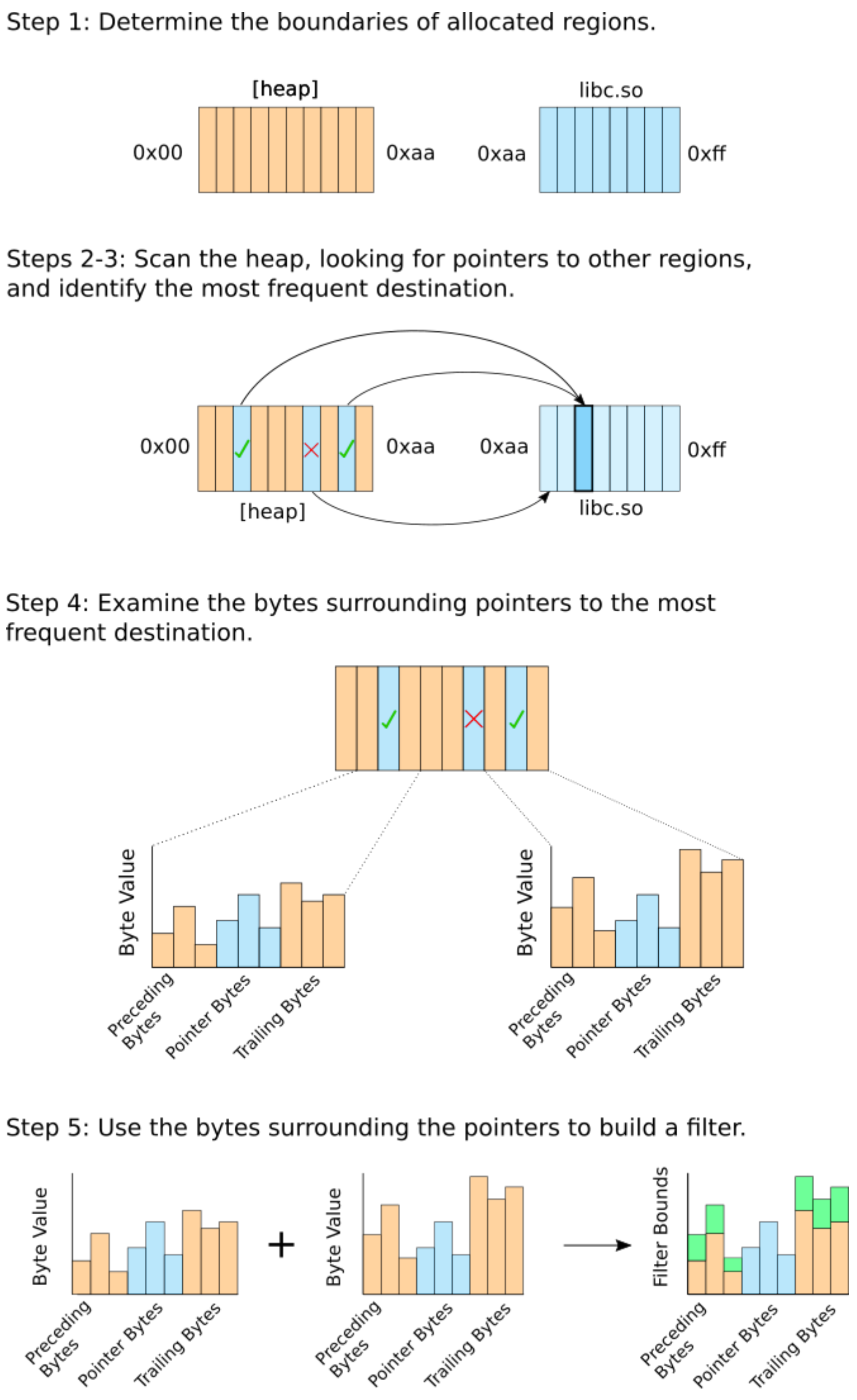

Fig. 1: The algorithm for building a pointer signature. In this example, three potential pointers are identified; those pointers point to two distinct destinations in `libc.so`.

One of these tuples corresponds to the target heap (i.e., the heap which the attacker can inspect using a read vulnerability). In some cases (like when the victim program uses a `mmap`-based allocator), the target heap may be spread over multiple VMAs. In these instances, the attacker analyzes the contents of all VMAs that comprise the target heap.

- **Step 2: Harvest Pointers.** The attacker begins scanning the target heap and analyzing every pointer-sized region of memory. Suppose an 8-byte region of memory on a 64-bit machine contains a value `val` such that

  `region_start <= val < region_end`

  for some allocated region of memory. The attacker then adds the tuple

  `(region_name, val - region_start)`

  to a list of observed pointer destinations.

- **Step 3: Frequency Analysis.** The attacker repeats the previous steps several times with independent runs of the program. After combining the recorded pointer destinations across all runs of the program, the attacker identifies which destinations are the most frequent. The attacker will prioritize the fingerprinting of pointers that reference the most frequent destinations; analyzing such pointers is likely to generate accurate fingerprints because such pointers are likely to be true pointers (instead of memory locations whose contents occasionally happen to represent addresses in a valid memory range). In a real-world exploit, the attacker would start by fingerprinting pointers to the most frequent destination and would move on to less frequent destinations until they were able to generate a fingerprint with satisfactory performance. In this paper, we report the results of fingerprinting the four most frequent destinations that were observed across every program run. Note that we record the destination address as an offset from the base of its enclosing memory region. This allows us to identify pointers to the same destination across program runs despite ASLR, provided that the destination is located in a region with a deterministic memory layout. In our threat model, the attacker's goal is to identify pointers to static data sections, so the relevant pointer destinations will be located at consistent offsets as required.
- **Step 4: Fingerprinting.** After generating a collection of frequently referenced destinations, the attacker chooses one of these destinations to target. The attacker then iterates over a set of heap snapshots from each run of the program. Whenever they observe a pointer to the target destination, they record all of the bytes in a fixed-width window around the pointer. We heuristically chose a window width of 64 bytes after comparing multiple window sizes. Finally, the attacker examines all of the recorded windows and determines the minimum and maximum value for each byte position in the window. For example, if the window were 64 bytes long, the attacker would derive a list of 64 lower bounds and 64 upper bounds. If 64 sequential bytes from memory all fall within this sequence of bounds, then those 64 bytes may contain a pointer to the target destination. Note that the window includes bytes that precede the pointer, bytes that follow the pointer, and bytes that are part of the pointer itself. We chose to include the bytes of the pointer in our analysis because the pointer bytes which are not affected by ASLR can be informative. For bytes that are part of the pointer, the derived bounds are very loose for bytes that are subject to ALSR, and very tight for bytes that are not.
- **Step 5: Filtering.** During the online phase, the attacker exploits a read vulnerability in a victim program. The attacker reads bytes from the heap, looking for regions of memory that fall within the sequence of bounds determined by offline analysis. Once the attacker finds a match, they can follow the identified pointer to a known offset inside a known region of memory, and conduct further memory cartography to reach the ultimate target region.

IV. Experiments

We implemented the algorithm described above using GDB and Python: GDB dumped memory from target regions, which was then analyzed in a Python script. Our open-source implementation is available at **https://github.com/jproney/MemoryCartography**. After performing our offline analyses, we then launched attacks on several real-world programs. Our tests assumed the presence of a heap read vulnerability; since the focus of this paper is leveraging such vulnerabilities, not discovering new ones, we used GDB to perform the attack-time memory scan. The assumption that the attacker possesses a read vulnerability is reasonable, since read vulnerabilities are often used as the initial step in modern exploits. All tests, unless otherwise stated, were conducted on 64-bit Ubuntu Linux version 20.04. Future work should evaluate the effectiveness of our methods on other platforms; however, we do not expect significantly different performance, especially because Rogowski et al. previously demonstrated the efficacy of memory cartography on Windows and macOS [5].

A. Vim

Vim is an open-source command-line text editor which is written primarily in C [19]. We chose Vim as our initial test program because it is single-threaded and has a well-defined heap region, but is complex enough to have considerable nondeterminism in its heap allocations; in other words, Vim exhibits variation in the types, locations, and number of allocated objects. A program being single-threaded and having a well-defined heap is by no means necessary for our method to work, but such a program simplified initial testing since we could easily determine which thread and VMA to analyze. We examine more complex examples later in this section.

The results of our experiments on Vim are presented in Table I. We were able to accurately identify pointers to data sections by scanning the heap; these pointers referenced memory regions with high connectivity to the rest of Vim's address space, making them ideal starting points for a memory cartography attack. In Table 1 (as well as subsequent tables), we report the precision and recall of our method when identifying pointers to various destinations. It should be noted that precision is the most important metric we report, since it approximates the probability that an attacker will find and follow a legitimate pointer to the intended destination, rather than following a falsely-discovered pointer and triggering a memory error.

B. Mozilla Firefox

Mozilla Firefox is an open-source web browser with hundreds of millions of global users. Web browsers are some of the highest-profile targets for data-oriented exploits like memory cartography, since browsers store extremely sensitive

TABLE I: Results on Vim

Rank	Region	Offset	True Positives	False Positives	Precision	Recall
1	vim_basic_4	90912	25650 / 25650	0 / 1525680	1.0	1.0
2	libc-2.31.so_5	3040	25451 / 25452	160 / 3077198	.994	.999
12	libc-2.31.so_5	2816	2360 / 2361	1375 / 3100289	.632	.999
14	libc-2.31.so_5	3072	800 / 802	1378 / 3101848	.367	.998

The accuracy of identifying pointers to various destinations in Vim. The "Region" and "Offset" columns indicate the pointer destination. The numbers next to memory regions differentiate memory allocations with the same name. "Rank" refers to the destination's frequency rank among all identified destinations. The frequencies are 1-indexed, with rank 1 being the most frequent destination. In each cross-validation run, 9/10 runs were used to compute a filter, and the last run was used to assess performance. True positives are regions of memory from the holdout run that match the filter and contain a pointer to the target destination; false positives match the filter but do not contain the right pointer. The denominators in the "True Positives" and "False Positives" sections are the total number of aligned pointers to the destination and the total number of aligned pointer-sized regions which do not point to the destination, respectively. Note that the denominator in the "False Positives" section can vary considerably between destinations, since we only check addresses that match the alignment of the true pointers from the training data. Results in the table are aggregated across all 10 cross-validation runs. Table I depicts results for the four most frequent pointers that were identified in every run.

TABLE II: Results on Firefox

Rank	Region	Offset	True Positives	False Positives	Precision	Recall	Precision (worst region)
1	libxul.so_2	21438312	310724 / 310735	662 / 6242515	.998	.999	.988
2	libxul.so_2	21438264	299715 / 299716	755 / 6253534	.997	.999	.993
3	libxul.so_1	27080560	23704 / 25603	2369 / 1612697	.909	.926	0.0
4	libxul.so_1	27085200	17300 / 18850	38 / 800250	.998	.918	0.0

The accuracy of identifying pointers to various destinations in Firefox. These are the four most frequent pointers that were identified in every run. The precision on the worst region represents the lowest performance across any Firefox heap "chunk." If the pointer in question was not present in one or more heap chunks, this was treated as a worst-case precision of 0.0.

TABLE III: Results on Firefox: Typed Arrays

Rank	Region	Offset	True Positives	False Positives	Precision	Recall
1	libxul.so_3	58432	22 / 30	0 / 394	1.0	.733
2	libxul.so_3	58752	21 / 30	0 / 394	1.0	.700
3	libxul.so_3	58816	21 / 30	0 / 394	1.0	.700
4	libxul.so_2	248768	22 / 30	0 / 394	1.0	.733

The accuracy of identifying pointers to various destinations in Firefox, using a large `jemalloc` slab allocated for a typed array. These are the four most frequent pointers that were identified in every run. As explained in Section IV.B, the discovered pointers were not in the large slab itself, but in the immediately adjacent allocations.

information like cookies, passwords, and other user data. Additionally, browsers have hundreds of interacting memory regions, with JavaScript engines and HTML parsers exposing large attack surfaces.

1) Overview of Sensitive Data: To motivate our study of Mozilla Firefox, we manually searched through memory in Firefox's JavaScript execution process and looked for sensitive data that could be disclosed in a memory cartography attack. We found the following information:

- plaintext descriptions of user devices, such as mouse driver names;
- plaintext file paths for user configuration files, with filenames often directly embedding usernames;
- plaintext references to four out of six browser extensions used by the test browser;
- plaintext descriptions of the operating system theme.

The above data are unavailable to non-malicious JavaScript code. JavaScript can enumerate media devices but not input devices; JavaScript cannot see the location of the Firefox binary on disk; JavaScript cannot generally detect extensions unless the extension creates extension-specific namespace artifacts; and JavaScript can detect dark-mode but not finer details about the OS theme.

The above information is sensitive for two reasons. First, this personal information may enable more thorough fingerprinting of users [20]. Second, determining which extensions and device drivers a user has installed can allow malicious websites to target specific vulnerabilities.

Before the implementation of stronger process isolation in browsers, malicious websites could leverage memory cartography attacks to steal cookies and passwords for other websites [5]. While out-of-process iframes [21] have greatly mitigated the risk of such attacks, it is still possible that implementation bugs could leave these sensitive objects exposed in memory.

2) Locating Heap Memory: Firefox lacks a well-defined, contiguous heap region. The reason is that Firefox uses a heavily-modified fork of `jemalloc` to dynamically allocate memory. `jemalloc` partitions memory into "chunks," which are further divided into "runs." In Firefox, chunks are always 1MB in size, and are allocated by `jemalloc` using calls to `mmap` [22], [23]. As a result, the Firefox heap is spread between a number of chunks that are not necessarily contiguous. In our attacks, we assumed that all 1MB anonymous allocations within a Firefox renderer process represented heap chunks; we treated the union of these regions as the effective program heap. Using the memory-viewing capabilities of Firefox's built-in developer tools, we confirmed that, except for very large objects, all allocated JavaScript objects were located within 1MB anonymous memory regions. Thus, we were confident that the union of 1MB anonymous regions contained at least a significant subset of the heap space used by the JavaScript runtime environment. Recall that our threat model involves an attacker who lacks source code and debugging symbols; thus, we did not inspect Firefox's codebase for ground truth about what Firefox places in `jemalloc` chunks.

When the program heap is fragmented over multiple memory regions, we calculate precision and recall metrics by aggregating true and false positives over all heap fragments. In this context, precision represents the probability an attacker would be successful if all of the heap fragments were visible via a read vulnerability. We also report the worst precision across any heap fragment, which indicates the worst-case success probability if the attacker were restricted to viewing data from a single fragment. A high worst-case precision indicates that the attacker can reliably navigate to a specific data section destination regardless of which heap fragment is exposed to the read vulnerability.

We tested our ability to identify data-section pointers within the Firefox heap after loading the webpage `https://mozilla.com`. The results of our experiments on Firefox are reported in Table II. We identified pointers to `libxul.so` with high precision; this library is well connected to heaps and data sections throughout the address space.

If JavaScript allocates an object larger than 1MB, Firefox and `jemalloc` create a special heap region to host this object [24]. In addition to assessing the performance of our pointer-recognition algorithms on standard 1MB heap chunks, we also tested our ability to locate data-section pointers in these large heap regions. To do so we created a webpage that allocates a JavaScript `TypedArray` object with size 40MB. We then identified the region backing this object in `/proc/<pid>/maps` based on its size. We inspected this region, and we could not find any consistent pointers to data sections. Note that the absence of consistent pointers is different from false negatives generated by our fingerprinting method. In the process of building a fingerprint, we scan memory dumps from local program executions and look for destinations in external memory regions that are consistently referenced across multiple program runs. If no such destinations exist, then successful fingerprinting is impossible, since there are no useful pointers to be detected. While there were no usable pointers in the large heap region, we could successfully identify pointers to data sections if we assumed the attacker could scan immediately adjacent memory regions (i.e., regions which were contiguous with the large heap region, but which were allocated by a different call to `mmap` and listed separately in `/proc/<pid>/maps`). The results of our pointer-identification efforts on large heap segments (and adjacent regions) are presented in Table III. As in the smaller Firefox heap regions, we uncovered pointers to `libxul.so` with high precision.

C. Apache and WordPress

Next, we evaluated the efficacy of our methods on server-side programs; to the best of our knowledge, no prior work has examined the susceptibility of server-side programs to memory cartography attacks. We began by examining the Apache web server, which is among the most popular web server frameworks [25].

To simulate a realistic web server, we installed a standard LAMP (Linux, Apache, MySQL, PHP) stack and configured Apache to serve a small WordPress site over HTTPS. To simplify the process of mapping the server's memory layout, we configured Apache to spawn a single worker process with multiple threads to handle all connections. This is a realistic configuration for high-traffic servers that aim to limit the number of processes forked. To further increase the realism of our attack, we ran Apache with Ubuntu 12.04 and OpenSSL 1.0.1, which is vulnerable to the infamous HeartBleed bug. HeartBleed is a classic buffer overread, disclosing the memory surrounding a vulnerable buffer. Our simulated attacker used the heap area containing the vulnerable buffer as the starting region for the cartography attack.

We collected pointers from this region after launching Apache and simulating various kinds of requests to the server, including user commenting, the creation and deletion of posts, and admin logins and logouts. The results of our experiments on Apache are reported in Table IV. As in our previous experiments, we identified useful pointers with very high accuracy.

D. Redis

Redis is a popular in-memory database that can function as a message broker or cache. Because Redis is primarily in-memory and lives within a single process, memory cartography can reveal data belonging to arbitrary users. We chose to analyze memory regions containing key-value pairs, since a bug in Redis's data retrieval code could unintentionally leak bytes from these regions during database queries.

By default, Redis uses `jemalloc` for memory allocation. (Refer to Section IV.B for further discussion of this allocator). After executing a workload of multiple set-adds to a single set, we empirically observed that all key-value pairs were located in 8MB anonymous memory regions, and that all 8MB anonymous regions contained key-value pairs. We therefore

TABLE IV: Results on Apache

Rank	Region	Offset	True Positives	False Positives	Precision	Recall
1	libphp5.so_1	252140	48542 / 51565	84 / 3655165	.998	.941
2	libphp5.so_0	3119280	45109 / 45109	11 / 3661621	.999	1.0
3	libphp5.so_0	3100304	26020 / 26020	0 / 3680710	1.0	1.0
4	libphp5.so_0	3213931	21850 / 21850	0 / 3684880	1.0	1.0

The accuracy of identifying pointers to various destinations in Apache. These are the four most frequent pointers that were identified in every run. Pointers were extracted from the heap which leaked data through the HeartBleed vulnerability.

TABLE V: Results on Redis

Rank	Region	Offset	True Positives	False Positives	Precision	Recall	Precision (worst region)
1	redis-check-rdb_3	352	330 / 330	30 / 10485430	0.917	1.0	0.0
5	locale-archive_0	109100	50 / 50	0 / 41942990	1.0	1.0	0.0
8	liblua5.1.so.0.0.0_1	4064	40 / 40	4 / 41943000	0.910	1.0	0.0
9	libc-2.31.so_1	1036947	30 / 30	0 / 20971490	1.0	1.0	0.0

The accuracy of identifying pointers to various destinations in Redis. Pointers to anonymous regions are not included in rank indexing. These are the four most frequent pointers that were identified in every run.

decided to treat the union of all 8MB anonymous regions as the effective set of vulnerable heap memory.

We then attempted to identify pointers in these regions. The results of these experiments are reported in Table V. We found that, during every execution, a single 8MB region contained reliably identifiable pointers, while the other regions contained no consistent pointers. We hypothesize that the region with identifiable pointers functions as the root of a linked list of memory regions, and that each region in the list contains a subset of the database's key-value pairs.

V. Discussion

In all test cases, we found pointers that could be identified with over 99% precision. For Vim, Apache, and Redis, at least one pointer could be identified in new runs of the program with perfect precision. In Firefox, the most predictable pointers had high enough precision (>99% on the worst heap region) that an attacker would be overwhelmingly likely to identify a true pointer when performing an online scan of the heap. Only the two most frequent pointer destinations were identified in all of Firefox's 1MB heap chunks; this is not very surprising, since different parts of the heap may contain different types of data with pointers to different regions of memory. It should also be noted that we observed considerable nondeterminism in the number of pointers present inside heaps between runs (Figure 2), suggesting that our learned filters are robust to the inherent uncertainty of heap allocation.

Our results indicate that it is possible to reliably navigate to a data section from unstructured heap data. A logical next question is whether navigation to these data sections is sufficient to launch a powerful memory cartography attack. To answer this question, we built memory graphs for all of our test applications as described by Rogowski et al. [5]. In the memory graph, each vertex is an allocated memory region, and directed edges between regions represent pointers that consistently link a fixed offset in the source region with a fixed offset in the destination region. We identified strongly connected components (SCCs) within these graphs. Any node in an SCC is reachable from any other node.

- In Vim, we found that our ability to navigate to the `vim_basic` and `libc-2.31.so` data sections was sufficient to access nearly all mapped regions in the program's address space. More specifically, `vim_basic` and `libc-2.31.so` were part of the largest SCC, which contained a large subset of allocated regions and had outgoing edges to virtually all other SCCs.
- In Firefox, we were able to reliably reach `libxul.so` from the heap. `libxul.so` is a critical library for the Gecko browser engine, and it resides within the largest SCC of the memory graph.
- We discovered pointers to regions of similarly high connectivity in Apache and Redis, suggesting that our methods enable navigation to regions that are ideal for launching memory cartography attacks.

The ability to identify specific pointers on the heap may be useful for other types of attacks. For instance, if an attacker wanted to leak the address of a library like `libc`, we hypothesize that the attacker could learn to identify pointers to a specific function in the library. By locating such a pointer in the heap, and then reading the pointer's value, the attacker could subtract the constant offset of this function to leak the base address of `libc`, potentially enabling a variety of attacks (like return-oriented programming) that subvert the control flow of the program [3].

To mitigate the threat posed by memory cartography attacks, software must ensure that no potentially sensitive information is stored in the same process as a potentially compromised execution flow. Sensitive data should instead be stored in

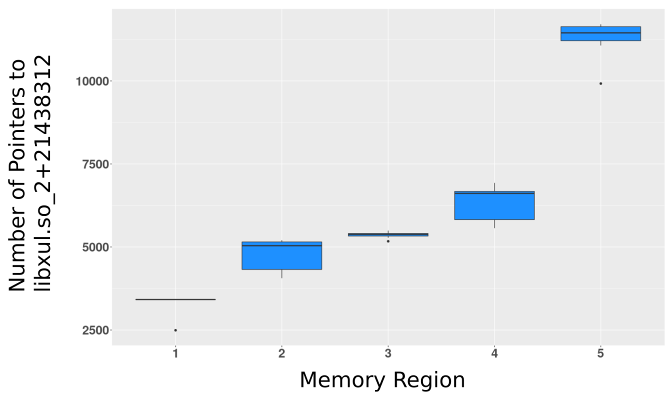

Fig. 2: Example of heap nondeterminism. Each "Memory Region" is a different 1MB anonymous region in the memory of Firefox. We ran Firefox 10 times and scanned each of these five 1MB regions, recording the number of pointers to `libxul.so_2`, offset 21438312 (this was the most frequently-referenced destination across all Firefox heaps). Each column contains data for a specific region, and displays the distribution of the number of pointers to `libxul.so_2+21438312` across all 10 runs.

another process, and then fetched by the unsafe process via IPC. This approach is already used by Chrome's site isolation architecture [21]. More sophisticated ASLR that randomizes more than just the offsets of memory regions might also decrease the precision of traversing process memory; however, some evidence suggests that fine-grained ASLR is less useful than one might expect [26].

VI. Limitations and Future Work

Our initial results are promising, but there are limitations to our current approach. For example, in some situations, our attack may be infeasible because the attacker does not have enough time to scan sufficient memory to find pointers; this might happen if the memory read vulnerability involves sending packets over a network. Additional research is also necessary to determine the extent to which the offline analysis phase must exactly predict the execution environment of the target program. For example, what if the offline analysis uses the same version of Chrome as the eventual victim, but uses Ubuntu 18.04 instead of 20.04? Our current experiments assume a perfect match, but the likelihood of attack success will presumably decrease as the victim environment looks "more different" than the test environment. As mentioned earlier, such changes could pose a challenge because the offsets of inter-region pointers within libraries may change between versions; if this happens, the attacker may try to follow a pointer at a known offset, only to trigger a memory error. Test-time versus attack-time environmental divergences are a classic challenge for attacks that exploit low-level execution characteristics.

Regardless, it is somewhat surprising that such a simple algorithm can identify specific pointers with such high accuracy. It would be interesting to explore the effectiveness of more complex learning algorithms like decision trees and SVMs. However, these algorithms may be susceptible to overfitting the training data, especially because we train on relatively few independent program runs.

In future work, we hope to profile the types of data that are referenced by high-frequency pointers on the heap. For example, it would be interesting to verify our hypothesis that pointers to virtual tables and library functions constitute a large portion of data-section pointers in heap regions. A systematic analysis of frequent pointer destinations may also provide insight into which types of memory regions are readily accessible from pointers in the heap.

VII. Conclusions

In this paper, we demonstrated the feasibility of using extremely simple filters to identify data-section pointers in nondeterministic heap data. Our techniques work in the presence of defenses like ASLR and enable attackers to transform a heap read vulnerability into a powerful memory cartography attack. The original cartography attack of Rogowski et al. required attackers to use techniques like heap spraying to understand the memory layout around a vulnerable object. In contrast, our work replaces this application-specific procedure with an *application-agnostic algorithm* for identifying data-section pointers in heap memory. These findings expand the number of programs that are vulnerable to memory cartography attacks, underscoring the need for different address spaces to hold content from mutually-distrusting origins.

References

[1] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," *BlueHatIL*, 2019.

[2] The HeartBleed Bug, accessed December 10, 2020. https://heartbleed.com.

[3] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *Transactions on Information and System Security*, vol. 15, no. 1, 2012.

[4] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Symposium on Information, Computer and Communications Security*, ACM, 2015, pp. 555–566.

[5] 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," in *European Symposium on Security and Privacy*, IEEE, 2017, pp. 366–381.

[6] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Security Symposium*, USENIX Association, 2005, p. 12.

[7] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *Symposium on Security and Privacy*, IEEE, 2016, pp. 969–986.

[8] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic generation of data-oriented exploits,” in *Security Symposium*, USENIX Association, 2015, pp. 177–192.

[9] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, “Block oriented programming: Automating data-only attacks,” in *Conference on Computer and Communications Security*, ACM, 2018, pp. 1868–1882.

[10] A. Baliga, V. Ganapathy, and L. Iftode, “Automatic inference and enforcement of kernel data structure invariants,” in *Annual Computer Security Applications Conference*, IEEE, 2008, pp. 77–86.

[11] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, “Mapping kernel objects to enable systematic integrity checking,” in *Conference on Computer and Communications Security*, ACM, 2009, pp. 555–565.

[12] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. T. Giffin, “Robust signatures for kernel data structures,” in *Conference on Computer and Communications Security*, ACM, 2009, pp. 566–577.

[13] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, “Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures,” in *Network and Distributed System Security Symposium*, The Internet Society, 2011.

[14] Q. Feng, A. Prakash, H. Yin, and Z. Lin, “Mace: High-coverage and robust memory analysis for commodity operating systems,” in *Computer Security Applications Conference*, ACM, 2014.

[15] M. Graziano, A. Lanzi, and D. Balzarotti, “Hypervisor memory forensics,” in *International Symposium on Research in Attacks, Intrusions and Defenses*, Springer, 2013, pp. 21–40.

[16] I. Haller, A. Slowinska, and H. Bos, “Mempick: High-level data structure detection in c/c++ binaries,” in *Working Conference on Reverse Engineering*, IEEE, 2013, pp. 32–41.

[17] A. Cozzie, F. Stratton, H. Xue, and S. T. King, “Digging for data structures,” in *Symposium on Operating Systems Design and Implementation*, USENIX Association, 2008, pp. 255–266.

[18] M. Morton, J. Werner, P. Kintis, K. Z. Snow, M. Antonakakis, M. Polychronakis, and F. Monrose, “Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks,” in *European Symposium on Security and Privacy*, IEEE, 2018, pp. 167–182.

[19] Vim, accessed December 10, 2020. https://www.vim.org.

[20] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, “Browser fingerprinting: A survey,” *ACM Transactions on the Web*, vol. 14, no. 2, Apr. 2020.

[21] C. Reis, A. Moshchuk, and N. Oskov, “Site isolation: Process separation for web sites within the browser,” in *Security Symposium*, USENIX Association, 2019, pp. 1661–1678.

[22] P. Argyroudis and C. Karamitas, “Exploiting the jemalloc memory allocator: Owning firefox’s heap,” *Blackhat USA*, 2012.

[23] MDN Web Docs: Garbage Collection, accessed December 11, 2020. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Garbage_collection.

[24] Jemalloc, accessed December 11, 2020. http://jemalloc.net.

[25] Apache, accessed January 20, 2021. https://httpd.apache.org.

[26] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *Symposium on Security and Privacy*, IEEE, 2013, pp. 574–588.