2016 IEEE Symposium on Security and Privacy

Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector

Erik Bosman
Vrije Universiteit Amsterdam
erik@minemu.org

Kaveh Razavi
Vrije Universiteit Amsterdam
kaveh@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

***Abstract*—Memory deduplication, a well-known technique to reduce the memory footprint *across* virtual machines, is now also a default-on feature *inside* the Windows 8.1 and Windows 10 operating systems. Deduplication maps multiple identical copies of a physical page onto a single shared copy with copy-on-write semantics. As a result, a write to such a shared page triggers a page fault and is thus measurably slower than a write to a normal page. Prior work has shown that an attacker able to craft pages on the target system can use this timing difference as a simple single-bit side channel to discover that certain pages exist in the system.**

In this paper, we demonstrate that the deduplication side channel is much more powerful than previously assumed, potentially providing an attacker with a *weird machine* to read arbitrary data in the system. We first show that an attacker controlling the alignment and reuse of data in memory is able to perform byte-by-byte disclosure of sensitive data (such as randomized 64 bit pointers). Next, even without control over data alignment or reuse, we show that an attacker can still disclose high-entropy randomized pointers using a *birthday attack*. To show these primitives are practical, we present an end-to-end JavaScript-based attack against the new Microsoft Edge browser, in absence of software bugs and with all defenses turned on. Our attack combines our deduplication-based primitives with a reliable Rowhammer exploit to gain arbitrary memory read and write access in the browser.

We conclude by extending our JavaScript-based attack to cross-process system-wide exploitation (using the popular nginx web server as an example) and discussing mitigation strategies.

I. Introduction

Memory deduplication is a popular technique to reduce the memory footprint of a running system by merging memory pages with the same contents. Until recently, its primary use was in virtualization solutions, allowing providers to host more virtual machines with the same amount of physical memory [32], [34], [7]. The last five years, however, have witnessed an increasingly widespread use of memory deduplication, with Windows 8.1 (and later versions) adopting it as a default feature *inside* the operating system itself [6].

After identifying a set of identical pages across one or more processes, the deduplication system creates a single read-only copy to be shared by all the processes in the group. The processes can freely perform read operations on the shared page, but any memory write results in a (copy-on-write) page fault creating a private copy for the writing process. Such write operation takes significantly longer than a write into a non-deduplicated page. This provides an attacker able to craft pages on the system with a single-bit timing side channel to identify whether a page with given content exists in the system. When using this simple side channel for information disclosure, the memory requirements grow exponentially with the number of target bits in a page, resulting in a very slow primitive which prior work argued useful only to leak low-entropy information [8].

In this paper, we show that memory deduplication can actually provide much stronger attack primitives than previously assumed, enabling an attacker to potentially disclose arbitrary data from memory. In particular, we show that deduplication-based primitives allow an attacker to leak even high-entropy sensitive data such as randomized 64 bit pointers and start off advanced exploitation campaigns. To substantiate our claims, we show that a JavaScript-enabled attacker can use our primitives to craft a reliable exploit based on the widespread Rowhammer hardware vulnerability [23]. We show that our exploit can allow an attacker to gain arbitrary memory read/write access and "own" a modern Microsoft Edge browser, even when the target browser is entirely free of bugs with all its defenses are turned on.

All our primitives exploit the key intuition that, if an attacker has some degree of control over the memory layout, she can dramatically amplify the strength of the memory deduplication side channel and reduce its memory requirements. In particular, we first show how control over the alignment of data in memory allows an attacker to pad sensitive information with known content. We use this padding primitive in conjunction with memory deduplication to perform byte-by-byte disclosure of high-entropy sensitive data such as *randomized code pointers*. We then extend this attack to situations where the target sensitive information has strong alignment properties. We show that, when memory is predictably reused (e.g., when using a locality-friendly memory allocator), an attacker can still perform byte-by-byte disclosure via partial data overwrites. Finally, we show that, even when entropy-reducing primitives based on controlled memory alignment or reuse are not viable, an attacker who can lure the target process into creating many specially crafted and interlinked pages can still rely on a sophisticated *birthday attack* to reduce the entropy and disclose high-entropy data such as *randomized heap pointers*.

© 2016, Erik Bosman. Under license to IEEE.
DOI 10.1109/SP.2016.63

IEEE computer society

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

After showcasing our deduplication-based primitives in a (Microsoft Edge) browser setting, we generalize our attacks to system-wide exploitation. We show that JavaScript-enabled attackers can break out of the browser sandbox and use our primitives on any other independent process (e.g., network service) running on the same system. As an example, we use our primitives to leak *HTTP password hashes* and break *heap ASLR* for the popular nginx web server.

To conclude our analysis, we present a Microsoft Edge case study which suggests that limiting the deduplication system to only *zero pages* can retain significant memory saving benefits, while hindering the attacker's ability to use our primitives for exploitation purposes in practice.

Summarizing, we make the following contributions:

- We describe novel memory deduplication-based primitives to create a programming abstraction (or *weird machine* [14], [37]) that can be used by an attacker to disclose sensitive data and start off powerful attacks on a target deduplication-enabled system (Section III).
- We describe an implementation of our memory deduplication-based primitives in JavaScript and evaluate their properties on the Microsoft Edge browser running on Windows 10 (Section IV and Section V).
- We employ our memory deduplication-based primitives to craft the first reliable Rowhammer exploit for the Microsoft Edge browser from JavaScript (Section VI).
- We show how our primitives can be extended to system-wide exploitation by exemplifying a JavaScript-based cross-process attack on the popular nginx web server running next to the browser (Section VII).
- We present a mitigation strategy (*zero-page deduplication*) that preserves substantial benefits of full memory deduplication (>80% in our case study) without making it programmable by an attacker (Section VIII).

II. Background

We first discuss the basic idea behind memory deduplication and its implementation on Windows and Linux. We then describe the traditional memory deduplication side channel explored in prior work and its limitations.

A. Memory Deduplication

To reduce the total memory footprint of a running system, memory pages with the same contents can be shared across independent processes. A well-known example of this optimization is the page cache in modern operating systems. The page cache stores a single cached copy of file system contents in memory and shares the copy across different processes. Memory deduplication generalizes this idea to the run-time memory footprint of running processes. Unlike the page cache, two or more pages with the same content are always deduplicated, even, in fact, if the pages are completely unrelated and their equivalence is fortuitous.

To keep a single copy of a number of identical pages, a memory deduplication system needs to perform three tasks:

1) Detect memory pages with the same content. This is usually done at regular and predetermined intervals during normal system operations [3], [6].
2) After detecting pages with the same content, keep only one physical copy and return the others to the memory allocator. For this purpose, the deduplication system updates the page-table entries (PTE) of the owning processes so that the virtual addresses (originally pointing to different pages with the same content) now point to a single shared copy. The PTEs are also marked as read-only to support copy-on-write (COW) semantics.
3) Create a private copy of the shared page whenever any process writes to it. Specifically, once one of owning processes writes to the read-only page, a (COW) page fault occurs. At this point, the memory deduplication system can create a private copy of the page and map it into the corresponding PTE of the faulting process.

On Windows (8.1 onward), memory deduplication is known as *memory combining*. The implementation merges pages that are both *private* and *pageable* [2] regardless of their permission bits. These pages exclude, for example, file-backed pages or huge pages which are non-pageable on Windows. To perform deduplication, memory combining relies on a kernel thread to scan the entire physical memory for pages with identical content. Every 15 minutes (by default), the thread calls the `MiCombineAllPhysicalMemory` function to merge all the identical memory pages found. On Linux, memory deduplication is known as *kernel samepage merging* (*KSM*). The implementation operates differently compared to Windows, combining both scanning and merging operations in periodic and incremental passes over physical memory [7].

B. The Memory Deduplication Side Channel

As mentioned earlier, writing to a shared page from any of the owning processes results in a page fault and a subsequent page copy. Due to these additional (expensive) operations, a write to a shared page takes significantly longer (up to one order of magnitude) compared to a write to a regular page.

This timing difference provides an attacker with a side channel to detect whether a given page exists in the system. For this purpose, she can craft a page with the exact same content, wait for some time, and then measure the time to perform a write operation to the crafted page. If the write takes longer than a write to a non-deduplicated page (e.g., a page with random content), the attacker concludes that a page with the same content exists. Using this capability, the attacker may be able to detect a user visiting a particular web page or running a particular program. We note that, while false positives here are possible (e.g., due to a non-unique crafted page or noisy events causing regular write operations to complete in unexpectedly long time), an attacker can

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

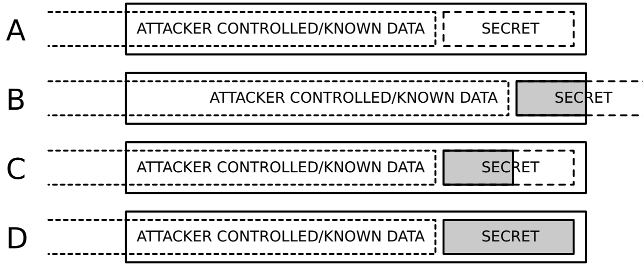

Figure 1. The *alignment probing* primitive to leak high-entropy secrets with weak memory alignment properties.

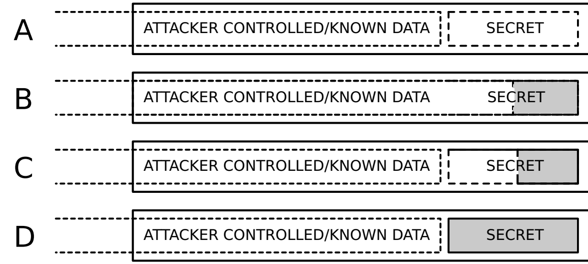

Figure 2. The *partial reuse* primitive to leak high-entropy secrets with predictable memory reuse properties.

use redundancy (e.g., repeated attempts or multiple crafted pages) to disclose the intended information in a reliable way.

At first glance, memory deduplication seems like a very slow single-bit side channel that can only be used for fingerprinting applications [31], [36] or at most leaking a limited number of bits from a victim process [8]. In the next section, we describe how memory deduplication can be abused to provide an attacker with much stronger primitives.

III. Attack Primitives

We describe efficient primitives based on the memory deduplication side channel to read *high-entropy* data from memory. Our primitives abuse a given deduplication system to build a weird machine [14], [37] that we can program by controlling the memory layout and generating pages with appropriate content. We later show that, by relying on our primitives, an attacker can program the weird machine to leak sensitive information such as randomized 64 bit pointers or even much larger secrets (e.g., 30 byte password hashes).

A naive strategy to disclose arbitrarily large secret information using the single-bit memory deduplication side channel is to brute force the space of all possible secret page instances. Brute forcing, however, requires the target page to be aligned in memory and imposes memory requirements which increase exponentially with each additional secret bit. This makes brute forcing high-entropy data not just extremely time consuming, but also unreliable due to the increasing possibility of false positives [8]. A more elegant solution is to disclose the target secret information incrementally or to rely on generative approaches. This intuition forms the basis for our memory deduplication-based primitives.

Primitive #1: Alignment probing

We craft a primitive that allows an attacker to perform byte-by-byte probing of secret information by controlling its *alignment*. Figure 1-A exemplifies a memory page with the secret data targeted by the attacker. We refer to pages that contain secret data as *secret pages* and to the pages that the attacker crafts to disclose the secret data as *probe pages*.

This primitive is applicable when attacker-controlled input can change the alignment of secret data with *weak alignment properties*. For instance, the secret may be a password stored in memory immediately after a blob of attacker-provided input bytes. By providing fewer or more bytes, the attacker can shift the password up and down in memory.

Using this capability, the attacker pushes the second part of the secret out of the target page (Figure 1-B), allowing her to brute force, using deduplication, only the first part of the secret (e.g., one byte) with much lower entropy. After obtaining the first part of the secret, the attacker provides a smaller input, so that the entire secret is now in one page (Figure 1-C). Next, she brute forces only the remainder of the secret to fully disclose the original data (Figure 1-D). With a larger secret, the attacker can simply change the alignment multiple times to incrementally disclose the data.

Our *alignment probing* primitive is very effective in practice. We later show how we used it to disclose a code pointer in Microsoft Edge and a password hash in nginx.

Primitive #2: Partial reuse

When the secret has strong memory alignment properties (e.g., randomized pointers), we cannot use our alignment probing primitive to reduce the entropy to a practical brute-force range. In this scenario, we craft a primitive that allows an attacker to perform byte-by-byte probing of secret information by controlling *partial reuse* patterns. This primitive is applicable when attacker-controlled input can partially overwrite stale secret data with *predictable reuse properties*.

User-space memory allocators encourage memory reuse and do not normally zero out deallocated buffers for performance reasons. This means that a target application often reuses the same memory page and selectively overwrites the content of a reused page with new data. If the application happens to have previously stored the target secret data in that page, the attacker can then overwrite part of the secret with known data and brute force the remainder.

Figure 2-A shows an example of a page that previously stored the secret and is reused to hold attacker-controlled input data. After partially overwriting the first part of the secret with a large input, the attacker can brute force, using deduplication, only the second part (Figure 2-B). Given the second part of the secret, the attacker can now brute force the first part by deduplicating against a page without an overwritten secret (Figure 2-C). Similar to the previous

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

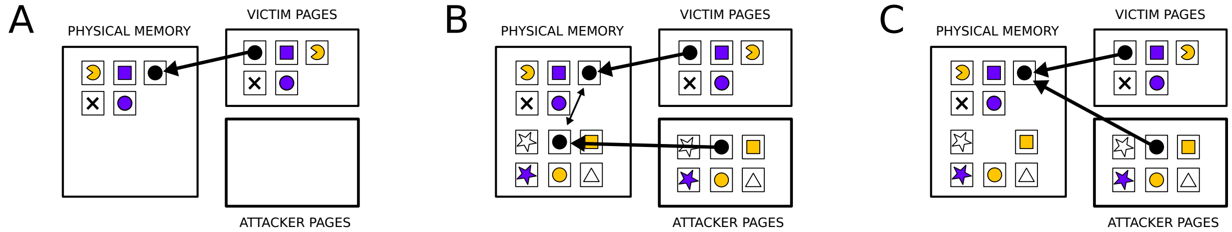

Figure 3. The *birthday heap spray* primitive to leak high-entropy heap ASLR with no attacker-controlled alignment or reuse.

primitive, the operations generalize to larger secrets and the result is full disclosure of the original data (Figure 2-D).

Our *partial reuse* primitive is fairly common in practice. We later show how we used it to break heap ASLR in nginx.

Primitive #3: Birthday heap spray

Our third primitive can leak a secret even when the attacker has no control over memory alignment or reuse. The primitive relies on a generative approach that revolves around the well-known birthday paradox, which states that the probability of at least two people in a group having the same birthday is high even for modest-sized groups. This primitive is applicable when the attacker can force the application to controllably *spray* target secrets over memory.

So far, we assumed that there is only one secret that we want to leak, so if a (partially masked) secret has P possible values, we use memory deduplication to perform $1 \times P$ comparisons between the P probe pages and the single target page—essentially brute forcing the secret. For a large P, doing so requires a prohibitively large amount of memory. In addition, it requires a large number of tests, which may lead to many false positives due to noise.

However, if the attacker can cause the target application to generate many secrets, memory deduplication provides a much stronger primitive than simple brute forcing. For instance, an attacker may generate a large number of (secret) heap pointers by creating a large number of objects from JavaScript, each referencing another object. For simplicity, we assume that the object is exactly one page in size and all fields are crafted constant and known except for one secret pointer, but other choices are possible. Whatever the page layout, its content serves as an *encoding* of the secret pointer.

Assume the attacker causes the application to generate S such pages, each with a different secret pointer (Figure 3-A). The attacker now also creates P probe pages, with P being roughly the same size as S. Each probe page uses the same encoding as the secret pages, except that, not knowing the secret pointers, the attacker needs to "guess" their values. Each probe page contains a different guessed value. The idea is to find at least one of the probe pages matching *any* of the secret pages. This is a classic birthday problem with the secret and probe values playing the role of birthdays.

Since memory deduplication compares any page with any other page in each deduplication pass, it automatically tests all our P possible probe pages against the S target secret pages (Figure 3-B). A hit on any of our P possible values immediately exposes a target secret (Figure 3-C).

Our birthday primitive reduces the memory requirements of the attack by a factor of S. It is especially useful when leaking the location of randomized pointers. Note that, for exploitation purposes, it is typically not important which pointer the attacker leaks, as long as at least one of them can be leaked. We later show how we used our primitive to leak a randomized heap pointer in Microsoft Edge and subsequently craft a reliable Rowhammer exploit.

IV. MICROSOFT EDGE INTERNALS

We discuss Microsoft Edge internals necessary to understand the attacks presented in Section V. First, we look at object allocation in Microsoft Edge's JavaScript engine, Chakra. We then describe how Chakra's JavaScript arrays of interest are represented natively. With these constructs, we show how an attacker can program memory deduplication from JavaScript. Finally, we describe how an attacker can reduce noise and reliably exploit our primitives.

A. Object Allocation

Chakra employs different allocation strategies for objects of different sizes maintained in three buckets [41]: small, medium, and large object buckets. The small and large object buckets are relevant and we discuss them in the following.

1) Small objects: Objects with a size between 1 and 768 bytes are allocated using a slab allocator. There are different pools for objects of different sizes in increments of 16 bytes. Each pool is four pages (16,384 bytes) in size and maintains contiguously allocated objects. In some cases, Chakra combines multiple, related allocations in one pool. This is the case, for example, for JavaScript arrays with a pre-allocated size of 17 elements or less, where an 88-byte header is allocated along with 17×8 bytes of data.

2) Large objects: Unlike small objects, large objects (i.e., larger than 8,192 bytes) are backed by a `HeapAlloc` call and are hence stored in a different memory location than their headers. One important consequence is that the elements of a large object have a known page alignment. As discussed in Section V-C2, we rely on this property to create our probe pages.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

B. Native Array Representation

Modern JavaScript has two different types of arrays. Regular *Arrays*, which can hold elements of any type and may even be used as a dictionary, and *TypedArrays* [29], which can only hold numerical values of a single type, have a fixed size, and cannot be used as a dictionary. TypedArrays are always backed by an *ArrayBuffer* object, which contiguously stores numerical elements using their native representation. Large ArrayBuffers are page-aligned by construction. To store regular Arrays, Chakra internally relies on several different representations. We focus here on the two representations used in our exploit.

The first representation can only be used for arrays which contain only numbers and are not used as dictionaries. With this representation, all the elements are sequentially encoded as double-precision IEEE754 floating-point numbers. This representation allows an attacker to create fake objects in the data part of the array. In particular, both pointers and small numbers can be encoded as denormalized doubles.

A second representation is used when an array may contain objects, strings, or arrays. For this representation, Microsoft Edge intelligently relies on the fact that double-precision IEEE754 floats have 2^{52} different ways of encoding both $+NaN$ and $-NaN$[1]. 2^{52} is sufficient to encode single values for $+NaN$ and $-NaN$, as well as 48 bit pointers and 32 bit integers. This is done by XORing the binary representation of doubles with `0xfffc000000000000` before storing them in the array. The 12 most significant bits of a double consist of a single sign bit and an 11-bit exponent. If the exponent bits are all ones, the number represents $+NaN$ or $-NaN$ (depending on the sign bit). The remaining 52 bits do not matter in JavaScript. As mentioned, Chakra only uses single values for $+NaN$ and $-NaN$, `0x7ff8000000000000` and `0xfff8000000000000` respectively. Since a user-space memory address has at least its 17 most significant bits set to 0, no double value overlaps with pointers by construction and Chakra can distinguish between the two cases without maintaining explicit type information.

The JIT compiler determines which representation to use based on heuristics. If the heuristics decide incorrectly (e.g., a string is later inserted into an array which can only contain doubles), the representation is changed in-place.

In Section V-C2, we show how we used these representations to craft our birthday heap spray primitive. Further, we used the details of the second representation to improve the success rate of our Rowhammer attack in Section VI.

C. Programming Memory Deduplication from JavaScript

To program memory deduplication, we need to be able to (a) create arbitrary pages in memory and (b) identify memory pages that have been successfully deduplicated. We use TypedArrays to create arbitrary memory pages and an available high-resolution timer in JavaScript to measure slow writes to deduplicated pages.

1) Crafting arbitrary memory pages: As mentioned in Section IV-B, TypedArrays can store native data types in memory. If the TypedArray is large enough, then the array is page-aligned and the location of each element in the page is known. Using, for example, a large Uint8Array, we can control the content of each byte at each offset within a memory page, allowing us to craft arbitrary memory pages.

2) Detecting deduplicated pages: While JavaScript provides no access to native timestamp counters via the `RDTSC` instruction, we can still rely on JavaScript's `performance.now()` to gather timing measurements with a resolution of hundreds of nanoseconds.

We detect a deduplicated page by measuring lengthy COW page faults when writing to the page. We measured that writing to a deduplicated page takes around four times longer than a regular page—including calls to `performance.now()`. This timing difference in conjunction with the ability to craft arbitrary memory pages provides us with a robust side channel in Microsoft Edge.

3) Detecting deduplication passes: As previously discussed in Section II, Windows calls `MiCombineAllPhysicalMemory` every 15 minutes to deduplicate pages. To detect when a deduplication pass occurs, we create pairs of pages with unique content, and write to pages belonging to different pairs every 10 seconds. Once a number of writes take considerably longer than a moving average, we conclude that a deduplication pass has occurred.

D. Dealing with Noise

To minimize the noise during our measurements, we used a number of techniques that we briefly describe here.

The first technique is to avoid cold caches. We first read from the address on which we are going to perform a write to ensure the page has not been swapped out to stable storage. Further, we call `performance.now()` a few times before doing the actual measurements in order to ensure its code is present in the CPU cache.

The second technique is to avoid interferences from the garbage collector (GC). We try to trigger the GC before doing any measurements. This can be done on most browsers by allocating and freeing small chunks of memory and detecting a sudden slowdown during allocations. On Microsoft Edge, however, it is possible to make a call to `window.CollectGarbage()` to directly invoke the GC.

The third technique is to avoid interferences from CPU's dynamic frequency scaling (DFS). We try to minimize noise from DFS by keeping the CPU in a busy loop for a few hundred milliseconds and ensuring that it is operating at the maximum frequency during our measurements.

Equipped with reliable timing and memory management

[1]Not a Number or NaN represents undefined values.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

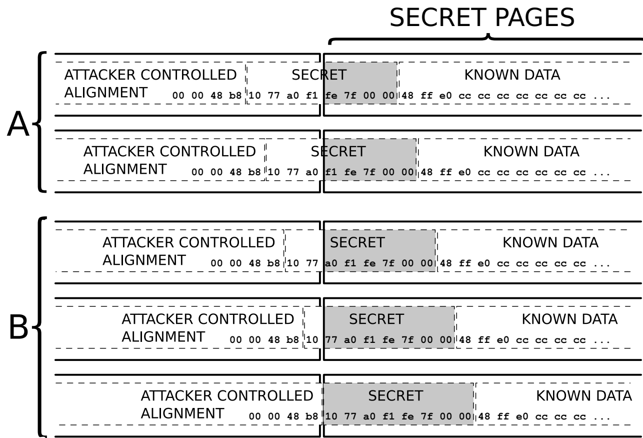

Figure 4. The incremental disclosure of a code pointer through JIT code. In the first deduplication pass, we can leak the higher bits of a randomized code pointer (A) and, in the second deduplication pass, we can leak the lower bits (B).

capabilities in JavaScript, we now move on to the implementation of our primitives and our Rowhammer attack.

V. Implementation

We now discuss the implementation details of the memory deduplication-based primitives introduced in Section III. Our implementation is written entirely in JavaScript and evaluated on Microsoft Edge running on Windows 10. We chose Microsoft Edge as our target platform since it is a modern browser that is designed from the ground up with security in mind. At the time of writing, Microsoft Edge is the only browser on Windows that ships as a complete 64 bit executable by default. 64 bit executables on Windows benefit from additional ASLR entropy compared to their 32 bit counterparts [27]. Nonetheless, we now show that using memory deduplication, we can leak pointers into the heap as well as pointers into interesting code regions.

Before detailing our end-to-end implementation, we first describe the testbed we used to develop our attacks.

A. Testbed

We used a PC with an MSI Z87-G43 motherboard, an Intel Core i7-4770 CPU, and 8 GB of DDR3 RAM clocked at 1600MHz running Windows 10.0.10240.

B. Leaking Code Pointers in Edge

We used our alignment probing primitive to leak code pointers in Microsoft Edge. Like all modern browsers, Microsoft Edge employs a JIT compiler which compiles JavaScript to native code. The generated code is full of references to memory locations (i.e., both heap pointers and code pointers) and, since x86 opcodes vary in size, most of these pointers turn out to be unaligned. This is ideal for our alignment probing primitive. We can first craft a large JavaScript routine mostly filled with known instructions that do not reference any pointers. Then, right in the middle, we can cause the generation of an instruction that contains a single pointer, and surgically position this pointer right across two page boundaries. We can then incrementally leak parts of the pointer across multiple deduplication passes. Although, in principle, this strategy sounds simple, complications arise when we account for security defenses deployed by modern browsers.

Microsoft Edge and other browsers randomize the JIT code they generate as a mitigation against return-oriented programming (ROP). Otherwise, a deterministic code generation scheme would allow attackers to generate their own ROP gadgets at a known offset in the JIT code. Randomization techniques include using XOR to mask constant immediates with a random value and insertion of dummy opcodes in the middle of JIT code. These lead to the presence of significant randomness to leak pointers in the middle of a JavaScript function with the approach described earlier. Fortunately, randomization does not affect the end of the JIT code generated for a given JavaScript function.

At the very end of each compiled JavaScript routine, we can find some exception handling code. The last two instructions of this code load a code pointer into the `RAX` register and jump there. The remainder of the page is filled with `int 3` instructions (i.e., `0xcc` opcode). The code pointer always points to the same code address in `chakra.dll` and can therefore reveal the base address of this DLL. For this to work, we need to make the JIT compiler create a routine which is slightly larger than one page in size. We could then push the code address partially across this page boundary, and create a second page where all data is known except for this partial code pointer. By pushing this code pointer further over the page boundary, we can expose more and more entropy to the second page as shown in Figure 4. This provides us with the semantics we need for our alignment probing primitive to work.

The only problem we still need to overcome is that we do not fully control the size of the routine due to the random insertion of dummy instructions. Given that the entropy of dummy instruction insertion is relatively low for practical reasons, we solved this by simply making the compiler generate JIT code for a few hundred identical routines. This results in a few pages for each possible alignment of our code pointer across the page boundary. At least one of them will inevitably be the desired alignment to probe for.

1) Dealing with noise: Although we have a very small number of probe pages, false positives may still occur. A simple way to sift out false positives when probing for secret pages with k unknown bytes, is to probe for pages with fewer (e.g., $k-1$) unknown bytes as well. Recall that we spray the JIT area with the same function, but due to the introduced randomness by the JIT compiler, different (last) pages end up with different number of bytes from the pointers. Hence, in the final pass we can probe for secret

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

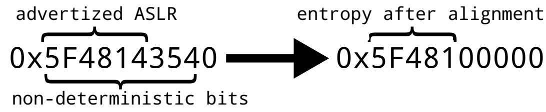

Figure 5. Entropy of an arbitrary randomized heap pointer before and after using the timing side channel.

pages which contain six, seven, and eight bytes all at once. Since the correct guesses contain the same information, they are redundant and can thus be used to verify each other using a simple voting scheme. Figure 4 shows how we exploit this redundancy to reduce the noise in the first and the second deduplication pass (A and B, respectively).

2) Time and memory requirements: ASLR entropy for 64 bit DLLs on Windows 10 is 19 bits. DLL mappings may start from `0x7ff800000000` to `0x7fffffff0000`. Assuming we know the exact version of `chakra.dll` (which allows us to predict the 16 least significant bits), we can easily leak the pointer's location in two deduplication passes. In the first pass, we can leak the five most significant bytes, as shown in Figure 4-A. Out of these bytes only the 11 least significant bits are unknown, hence, we only need 2^{11} probe pages requiring 8 MB of memory. In a second pass (Figure 4-B), we can probe for the remaining eight bits of entropy. Note that the memory requirement of this attack is orders of magnitude smaller than a sheer brute force, making this attack feasible in a browser setting.

In case the exact version of `chakra.dll` is unknown, we can opt to leak the pointer in three passes by leaking the two least significant bytes in the last pass. Assuming the code pointer is aligned on a 16 byte boundary, this requires 2^{12} probe pages and 16 MB of memory.

C. Leaking Heap Pointers in Edge

While we could incrementally leak randomized code pointers using our alignment probing primitive in Microsoft Edge, we did not find a similar scenario to leak randomized heap pointers given their strong alignment properties. To leak heap pointers, using our partial reuse primitive is an option, but given the strong security defenses against use-after-free vulnerabilities deployed in modern browsers, memory reuse is not easily predictable. Hence, a different strategy is preferable.

Since we have the ability to generate many heap pointers from JavaScript, we can craft our birthday heap spray primitive (Section III) to leak heap pointers. For this attack, we allocate many small heap objects and try to leak the locations of some of them. Before employing our birthday heap spray primitive to break heap ASLR, we first describe an additional novel timing side channel to further reduce the entropy in a preliminary step.

1) Reducing ASLR entropy: At the time of writing, 64 bit heap memory on Windows 10 starts at addresses ranging between `0x100000000` and `0x10000000000`, and aligned

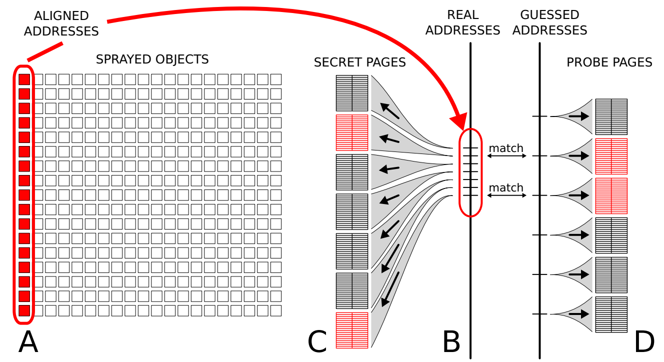

Figure 6. The birthday heap spray primitive to leak high-entropy heap ASLR with no attacker-controlled alignment or reuse. After finding the alignment of the sprayed heap objects via a side-channel (A), we encode the address of each 1 MB-aligned object (B) into a secret page by storing its reference multiple times (C). We then guess these secret pages by creating probe pages that mimic the layout of secret pages (D). In our current version, our guesses are 128 MB apart.

at 16-page boundaries. This leaves us with 24 bits of entropy for the heap, similar to what prior work previously reported for Windows 8 [27]. Most pointers used from JavaScript, however, do not align at 16-page boundaries and can point to any 16 byte-aligned location after their base offset (see Figure 5 for an example). This leaves us with 36 bits of entropy for randomized heap addresses.

Mounting our birthday heap spray primitive directly on 36 bits of entropy requires 2^{18} secret pages and the same number of probe pages, amounting to 1 GB of memory. Additionally, finding a signal in 2^{18} pages requires a very small false positive rate, which past research shows is difficult to achieve in practice [8]. However, using a timing side channel in Microsoft Edge's memory allocator, we can reliably detect objects that are aligned to 1 MB, reducing the entropy down to 20 bits.

Whenever Microsoft Edge's memory allocator runs out of memory, it needs to ask the operating system for new pages. Every time this happens, the allocator asks for 1 MB of memory. These allocations happen to also be aligned at the 1 MB boundary. In order to get many consecutive allocations, we spray many small array objects (see Section IV-A) on the heap in a tight loop and keep a reference to them so that they are not garbage collected. We time each of these allocations and mark the ones that take longer than eight times the average to complete. If, for example, it takes 4,992 objects to fill a 1 MB slab buffer, we try to find chains of slower allocations that are 4,992 allocations apart. These are the array objects aligned at the 1 MB boundary.

There may still be a small number of false positives in our (large) set of candidates. This is not an issue, as these candidates will simply not match any of the probe pages that we craft for our birthday heap spray primitive.

2) Birthday heap spray: Figure 6 summarizes the steps we followed to implement our birthday heap spray primitive.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

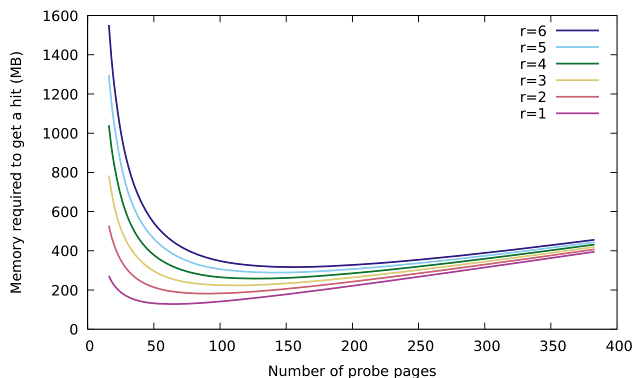

Figure 7. Birthday heap spray's reliability and memory tradeoffs.

As a first step, we allocate a number of target objects aligned at the 1 MB boundary using the timing side channel introduced earlier (Figure 6-A). The *reference* to a 1 MB-aligned object constitutes a secret that we want to leak using our birthday heap spray primitive. Since we can create an arbitrarily large number of such objects, we can force Microsoft Edge to generate S secrets for our birthday attack.

We encode each of the S secrets in a single secret page which we describe now. Our S secret pages are backed using a large Array, as described in Section IV-A. We fill in the array elements (at known offsets) with references to exactly one secret per page. Using this strategy, we can force each secret page to store 512 identical 64 bit pointers. Note that probing directly for a target object's header (without using our secret pages) incurs more entropy, since the header contains different pointers. This increases the entropy of the page that stores an object's header significantly. When using secret pages, in contrast, the only entropy originates from a single pointer referencing one of the S secrets (Figure 6-B/C).

We now need to craft P probe values to guess at least one of the S secrets. We encode each value in a probe page to mimic the layout of the secret pages. To store our probe pages, we create a large TypedArray, which, compared to a regular Array, offers more controllable value representations. We fill each array page with 512 guessed 64 bit pointers similar to the secret pages (Figure 6-D).

After the generation step completes, a few of our P probe pages get inevitably deduplicated with some of our S secret pages. Since we can detect deduplicated probe pages using our memory deduplication side channel, we can now leak correctly guessed pointer values and break heap ASLR.

3) Dealing with noise: If we want to add some redundancy in detecting an object's address, we cannot simply create extra identical probe pages. Identical probe pages get deduplicated together, resulting in false positives. However, since we have no restrictions on how to "encode" our secret into a page, we can add variations to create extra sets of secret and probe pages. One way is to fill all but one of the available slots (i.e., 511 slots) with a reference to our object, and fill the remaining slot with a different magic number for each redundant set of pages.

4) Time and memory requirements: Our implementation of the birthday heap spray primitive requires only a single deduplication pass to obtain a heap pointer. For the execution of the attack, we need to allocate three chunks of memory. A first chunk is needed for the S 1 MB-aligned target objects, resulting in $S \cdot 2^{20}$ bytes. A second chunk of memory is needed for the secret pages. With a redundancy factor r, we need $S \cdot r \cdot 2^{12}$ bytes for the secret pages. To cover 20 bits of entropy, we need $P = \frac{2^{20}}{S}$ probe pages, each with a r redundancy factor, resulting in $\frac{2^{20}}{S} \cdot r \cdot 2^{12}$ bytes. Figure 7 shows the memory requirements for different redundancy factors based on this formula. With our target redundancy factor of three (which we found sufficient and even conservative in practice), we can leak a heap pointer with only 500 MB of memory. Table I summarizes the end-to-end requirements for our attacks to leak code and heap pointers. Note that we can leak part of a code pointer and a complete heap pointer in the same deduplication pass (15 minutes). Given a known version of `chakra.dll`, we can leak both pointers in two deduplication passes (30 minutes).

Pointer type	Memory	Dedup passes	Time
Unknown code	16 MB	3	45 Minutes
Known code	8 MB	2	30 Minutes
Heap	500 MB	1	15 Minutes
Heap + unknown code	516 MB	3	45 Minutes
Heap + known code	508 MB	2	30 Minutes

Table I
TIME AND MEMORY REQUIREMENTS TO LEAK POINTERS IN THE CURRENT IMPLEMENTATION.

D. Discussion

In this section, we described the implementation of our memory deduplication primitives in Microsoft Edge's JavaScript engine. Using our alignment probing primitive, we leaked a randomized code pointer and, using our birthday heap spray primitive, we leaked a randomized heap pointer. We successfully repeated each of these attacks 10 times.

In the next section, we describe the first remote Rowhammer exploit that extensively relies on our memory deduplication primitives to disclose randomized pointers. To the best of our knowledge, ours is the first modern browser exploit that does not rely on any software vulnerability.

We did not need to employ the partial reuse primitive for our Rowhammer exploit, but, while we found that controlling memory reuse on the browser heap is nontrivial, we believe that our partial reuse primitive can be still used to leak randomized stack addresses by triggering deep functions in JIT code and partially overwriting the stack. In

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

Section VII, in turn, we extensively use our partial reuse primitive to leak a 30 byte password hash from a network server—part of a class of applications which is, in contrast, particularly susceptible to controlled memory reuse attacks.

VI. Rowhammering Microsoft Edge

Rowhammer [23] is a widespread DRAM vulnerability that allows an attacker to flip bits in a (victim) memory page by repeatedly reading from other (aggressor) memory pages. More precisely, repeated activations of rows of physical memory (due to repeated memory read operations) trigger the vulnerability. The bit flips are deterministic: once we identify a vulnerable memory location, it is possible to reproduce the bit flip patterns by reading again the same set of aggressor pages.

We report on the first reliable remote exploit for the Rowhammer vulnerability running entirely in Microsoft Edge. The exploit does not rely on any software vulnerability for reliable exploitation. It only relies on our alignment probing primitive and our birthday heap spray primitive to leak code and heap pointers (respectively), which we later use to create a counterfeit object. Our counterfeit object provides an attacker with read/write access to Microsoft Edge's virtual memory address space.

To reliably craft our end-to-end exploit, we had to overcome several challenges, which we now detail in the remainder of the section. First, we describe how we triggered the Rowhammer vulnerability in Microsoft Edge running on Windows 10. Next, we describe how we used our deduplication primitives to craft a (large) counterfeit JavaScript object inside the data area of a valid (small) target object. Finally, we describe how we used Rowhammer to pivot from a reference to a valid target object to our counterfeit object, resulting in arbitrary memory read/write capabilities in Microsoft Edge.

A. Rowhammer Variations

In the literature, there are two main variations on the Rowhammer attack. Single-sided Rowhammer repeatedly activates a single row to corrupt its neighbouring rows' cells. Double-sided Rowhammer targets a single row by repeatedly activating both its neighbouring rows. Prior research shows that double-sided Rowhammer is generally more effective than single-sided Rowhammer [33].

The authors of Rowhammer.js [17], an implementation of the Rowhammer attack in JavaScript, rely on Linux' anonymous huge page support. A huge page in a default Linux installation is allocated using 2 MB of contiguous physical memory—which spans across multiple DRAM rows. Hence, huge pages make it possible to perform double-sided Rowhammer from JavaScript. Unfortunately, we cannot rely on huge pages in our attack since Microsoft Edge does not explicitly request huge pages from the Windows kernel.

Another option we considered was to rely on large allocations. We expected Windows to allocate contiguous blocks of physical memory when requesting large amounts of memory from JavaScript. However, Windows hands out pages from multiple memory pools in a round robin fashion. The memory pages in each of these pools belong to the same CPU cache set [29], which means that large allocations are not backed by contiguous physical pages. We later made use of this observation to efficiently create cache eviction sets, but it is not immediately clear how we could use these memory pools to find memory pages that belong to adjunct memory rows and perform double-sided Rowhammer. Hence, we ultimately opted for single-sided Rowhammer in JavaScript.

B. Finding a Cache Eviction Set on Windows

The most effective way to hammer a row is to use the `clflush` instruction, which allows one to keep reading from main memory instead of the CPU cache. Another option is to find eviction sets for a specific memory location and exploit them to bypass the cache.

Since the `clflush` instruction is not available in JavaScript, we need to rely on eviction sets to perform Rowhammer. Earlier, we discovered that Windows hands out physical pages based on the underlying cache sets. As a result, the addresses that are 128 KB apart are often in the same cache set. We use this property to quickly find cache eviction sets for memory locations that we intend to hammer. Modern Intel processors after Sandy Bridge introduced a complex hash function to further partition the cache into slices [19], [26]. An address belongs to an eviction set if the address and the eviction set belong to the same cache slice. We use a cache reduction algorithm similar to [17] to find minimal eviction sets[2] in a fraction of a second.

To prevent our test code from interfering with our cache sets, we created two identical routines to perform Rowhammer and determine cache sets. The routines are placed one page apart in memory, which ensures the two routines are located on different cache sets. If one of the two routines interferes with an eviction set, the other one does not. In order to find our eviction set, we run our test on both routines and pick the fastest result. Likewise, before hammering, the fastest test run determines which routine to use.

C. Finding Bit Flips

In order to find a vulnerable memory location, we allocate a large Array filled with doubles. We make sure these doubles are encoded using the XOR pattern described in Section IV-B by explicitly placing some references in the Array. This allows us to encode a double value such that all bits are set to 1[3]. We then find eviction sets and hammer 32 pages at a time. We read from each page two million times

[2]Minimal eviction sets contain $N + 1$ entries for an N-way L3 cache.

[3]The double value 5.5626846462679985e-309 has all bits set considering the XOR pattern discussed in Section IV-B.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

before moving to the next page. After hammering each set of 32 pages, we check the entire Array for bit flips.

After scanning a sufficient number of pages, we know which bits can be flipped at which offsets. Next, we need to determine what to place in the vulnerable memory locations to craft our exploit. For this purpose, our goal is to place some data in our Array which, after a bit flip, can yield a reference to a controlled counterfeit object. We now first describe how to obtain a bit-flipped reference to a counterfeit object and then how to craft a counterfeit object to achieve arbitrary memory read/write capabilities.

D. Exploiting Bit Flips

We developed two possible techniques to exploit bit flips in Microsoft Edge:

1) Type flipping: Given the double representation described in Section IV-B, a high-to-low (i.e., 1-to-0) bit flip in the 11-bit exponent of a double element in our large Array allows us to craft a reference to any address, including that of our counterfeit object's header. In essence, this bit flip changes an attacker-controlled double number into a reference that points to an attacker-crafted counterfeit object.

2) Pivoting: Another option is to directly corrupt an existing valid reference. For this purpose, we can store a reference to a valid target object in a vulnerable array location. By corrupting the lower bits of the reference, we can then pivot to our counterfeit object's header. Assuming an exploitable high-to-low bit flip, our corrupted reference will point to a lower location in memory. If we fabricate our counterfeit object's header at this location, we can then use the corrupted reference to access any memory addressable by the counterfeit object. Recall from Section V-C1 that we sprayed our target objects close to each other. By corrupting a reference to one of these (small) objects, we obtain a reference to the middle of the previous valid target object. Since we control the memory contents pointed by the corrupted reference (our small target objects use in-band data), we can fabricate our counterfeit object at that location.

These two attacks make it possible to exploit 23 out of every 64 high-to-low bit flips (i.e., 36% of the bit flips are exploitable). We now describe how we create our counterfeit object before summarizing our attack.

E. Creating a Counterfeit JavaScript Object

To craft a valid large counterfeit object, we rely on the code pointer to the Chakra's binary we leaked using our alignment probing primitive, and on the heap pointer we leaked using our birthday heap spray primitive. Our counterfeit object (of type `Uint8Array`) resides inside a JavaScript Array containing only IEEE754 double values. As discussed in Section IV-B, this type of array does not XOR its values with a constant, allowing us to easily craft arbitrary pointers inside the array (or any other non-NaN value).

To craft our counterfeit `Uint8Array` object, we need a valid vtable pointer. We obtain the latter by simply adding a fixed offset to our leaked code pointer. Other important fields in the `Uint8Array` object are its size and a pointer to its out-of-band data buffer. We obtain the latter by simply using our leaked heap address. These fields are sufficient to allow a compiled JavaScript routine to use our `Uint8Array` object. The generated assembler performs a type comparison on the vtable pointer field and performs bound checking on the size field. Note that the crafted counterfeit object does not violate any of the CFI rules in Microsoft Edge [42].

At this stage, since we control the out-of-band data buffer location our counterfeit `Uint8Array` points to, we can read or write from anywhere in the Microsoft Edge's address space with valid virtual mappings. To reliably identify all the valid mappings, we can first use our counterfeit object to dump the contents of the current heap and find heap, stack or code pointers that disclose additional virtual mapping locations in the address space. We can now get access to the newly discovered locations by crafting additional counterfeit objects (using the current counterfeit object) and discover new pointers. Alternating disclosure steps with pointer chasing steps allows us to incrementally disclose the valid virtual mappings and control the entirety of the address space, as also shown in prior work [11], [13], [25], [35].

F. Dealing with Garbage Collection

Using the counterfeit objects directly (as done above) provides us with arbitrary read/write access to Microsoft Edge's address space, but as soon as a garbage collection pass checks our counterfeit object, the browser may crash due to inconsistent state in the garbage collector.

To avoid this scenario, we have to minimize the amount of time that we use the counterfeit object directly. For this purpose, *we only use the counterfeit object to corrupt the header of other valid objects* and we immediately restore the (corrupted) reference to our counterfeit object afterwards.

To this end, we use the (leaked) pointer to a valid target object's header as the backing data buffer of our crafted counterfeit `Uint8Array` object. This allows us to corrupt the size field of the target (array) object and operate out-of-bounds accesses from the corresponding valid array references. This again grants us arbitrary memory read/write access to the underlying heap and, using the incremental disclosure strategy described earlier, to the entirety of Microsoft Edge's address space.

G. Putting the Pieces Together

Using Figure 8, we now summarize the steps of our end-to-end Rowhammer attack with the pivoting technique described in Section VI-D. The attack using the type flipping technique is similar and we omit it for brevity.

As shown in Figure 8-A, at this stage of the attack, we have access to a bit flip inside a controlled array. We can

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

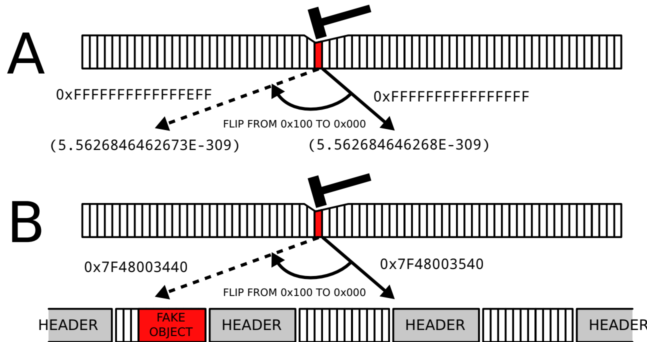

Figure 8. By flipping a bit in an object pointer, we can pivot to the attacker's counterfeit object. First, we identify a vulnerable memory location within an array (A). After finding an exploitable bit flip, we store a valid object reference at the vulnerable memory location and pivot to a counterfeit object with Rowhammer (B).

now trigger the bit flip and pivot to our counterfeit object. For this purpose, we store a reference to a valid object at the vulnerable location inside the large double array we created earlier (Section VI-C). We choose our valid object in a way that, when triggering a bit flip, its reference points to our counterfeit object, as shown in Figure 8-B.

With the arbitrary read/write primitive provided by our counterfeit object, gaining code execution is achievable even under a strong CFI implementation, as shown by [10].

1) Time and memory requirements: To leak the code and heap pointers necessary for our Rowhammer attack, we need 508 MB of memory and 30 minutes for two deduplication passes, as reported in Table I (assuming a known version of `chakra.dll`). In addition, for the Rowhammer attack, we need 1 GB of memory to find bit flips and 32 MB of memory for our cache eviction sets. The time to find an exploitable bit flip, finally, depends on the vulnerable DRAM chips considered, with prior large-scale studies reporting times ranging anywhere from seconds to hours in practice [23].

H. Discussion

In this section, we showed how an attacker can use our deduplication primitives to leak enough information from the browser and craft a reliable Rowhammer exploit. Our exploit does not rely on any software vulnerability and runs entirely in the browser, increasing its impact significantly. We later show how an in-browser attacker can use our primitives to also attack a process outside the browser sandbox and present mitigation strategies in Section VIII.

Finally, we note that, to trigger bit flips using Rowhammer, we had to increase our DRAM's refresh time, similar to Rowhammer.js [17]. However, we believe that more vulnerable DRAMs will readily be exploitable without modifying the default settings. We are currently investigating the possibility of double-sided Rowhammer in Microsoft Edge using additional side channels and more elaborate techniques to induce bit flips with the default DRAM settings.

VII. System-wide Exploitation

In the previous sections, we focused on a JavaScript-enabled attacker using our primitives to conduct an advanced exploitation campaign inside the browser. In this section, we show how the same attacker can break out of the browser sandbox and use our primitives for system-wide exploitation targeting unrelated processes on the same system. We focus our analysis on network servers, which accept untrusted input from the network and thus provide an attacker with an entry point to control memory alignment and reuse.

We consider an attacker running JavaScript in the browser and seeking to fulfill three system-wide goals: (i) fingerprinting the target network server version running on the same system; (ii) disclosing the password hash of the `admin` network server user; (iii) disclosing the heap (randomized using 64 bit ASLR) by leaking a heap pointer. We show that crafting our primitives to conduct all such attacks is remarkably easy for our attacker, despite the seemingly constrained attack environment. This is just by exploiting the strong spatial and temporal memory locality characteristics of typical high-performance network servers.

For our attacks, we use the popular nginx web server (v0.8.54) as an example. We use the 64 bit version of nginx running in Cygwin as a reference for simplicity, but beta nginx versions using the native Windows API are also available. We configure nginx with a single root-level (`\0`-terminated) password file containing a randomly generated HTTP password hash for the `admin` user and with 8 KB request-dedicated memory pools (`request_pool_size` configuration file directive). Using 8 KB pools (4 KB by default in our nginx version) is a plausible choice in practice, given that the maximum HTTP request header length is 8 KB and part of the header data is stored in a single pool. Before detailing the proposed deduplication-based attacks, we now briefly summarize nginx' memory allocation behavior.

A. Memory Allocation Behavior

nginx implements two custom memory allocators on top of the standard `malloc` implementation, a slab allocator and a region-based (or pool) allocator [9]. We focus our analysis here on nginx' pool allocator, given that it manages both user (and thus attacker-controllable) data and security-sensitive data, and also tends to allocate many small objects consecutively in memory.

The pool allocator maintains a number of independent pools, each containing logically and temporally related objects. Clients are provided with `create`, `alloc`, and `destroy` primitives to respectively create a new pool, allocate objects in it, and destroy an existing pool (with all the allocated objects). Each pool internally maintains two singly linked lists of data blocks (`ngx_pool_t`) and large blocks (`ngx_pool_large_t`). The latter are only used for large chunks (larger than 4 KB)—rarely allocated during regular execution—so we focus our analysis on the former.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

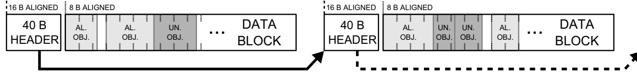

Figure 9. Aligned and unaligned objects allocated in a single nginx pool.

Figure 9 exemplifies how nginx manages data blocks in a single pool and (small) objects with the contained data blocks. Each data block is allocated through the standard `malloc` allocator, using a fixed 8 KB size and a 16 byte alignment. The data block is prepended with a 40 byte pool header, while the rest of the block is entirely dedicated to object allocations. Within a block, objects are allocated consecutively similar to a simple buffer allocator. For this purpose, the pool header maintains the pointer to the next free chunk in the corresponding data block.

The pool allocator offers two allocation primitives to allocate objects in each pool: `pnalloc` (which allocates *unaligned objects* at the byte boundary) and `palloc` (which allocates *aligned objects* at the 8 byte boundary). Whatever the allocation primitive, the allocator checks if there is space available for the requested object size in the current (i.e., last allocated) data block. If sufficient space is available, the object is allocated right after the last allocated object. Otherwise (provided no space is available in other data blocks), the allocator `mallocs` and appends a new data block to the pool's data block list. In such a case, the object is allocated in the new data block right after the header.

Note that, once allocated, an object (or a data block) lives until the pool is destroyed, as the allocator offers no object- or block-level deallocation primitives. This simple design is ideal to allocate logically related objects with a limited and well-determined lifetime. Among others, nginx maintains one pool for each connection (*connection pool*) and one pool for each HTTP request (*request pool*). The request pool, in particular, contains much client-controlled and security-sensitive data, which an attacker can abuse to fulfill its goals, as we show in Section VII-D and Section VII-E.

B. Controlling the Heap

To craft our primitives, an attacker needs to control the layout of data in memory and ensure that the target data are not overwritten before a deduplication pass occurs. While this is trivial to implement in the attacker-controlled memory area in a browser, it is slightly more complicated for server programs with many concurrent connections and allocators that promote memory reuse for efficiency.

Despite the challenges, we now show how an attacker can spray the heap of a network server such as nginx to reliably force the memory allocator to generate a long-lived page-sized data pattern in memory. The pattern contains attacker-controlled data followed by a target secret. In particular, we show an attacker can fulfill two goals: (i) achieving a target pattern alignment to retain a controlled number of secret bytes before the page boundary, and (ii) ensuring that the target pattern is not overwritten by other data.

To achieve our first goal, we can simply spray the heap using thousands of parallel HTTP requests, ensuring some of the generated patterns will land on a target alignment within a page with high probability. To verify this intuition, we issued 100,000 HTTP requests using 1,000 concurrent connections to nginx. Our results confirmed that we can easily gain 22 unique patterns with a given target alignment on the heap (median of 11 runs) in a matter of seconds.

To achieve our second goal, we need to prevent at least some of the pages hosting the generated pattern from being overwritten by other data. This is not possible within the same HTTP request (the request pool allows for no internal reuse by construction), but it is, in principle, possible after the server has finished handling the request. Since each data block is allocated through the standard `malloc` allocator, reuse patterns depend on the underlying system allocator.

Standard `malloc` allocators are based on free lists of memory blocks (e.g, ptmalloc [5]) and maintain per-size free lists in MRU (Most Recently Used) fashion. This strategy strongly encourages reuse patterns across blocks of the same size. Since the size of the data block in the request pool is unique in nginx during regular execution, this essentially translates to attacker-controlled request data blocks being likely only reused by other requests' data blocks. Some interference may occur with allocators coalescing neighboring free blocks, but the interference is low in practice, especially for common server programs such as nginx which use very few fixed and sparse allocation sizes.

Hence, the main question that we need to answer is whether patterns with the target alignment are overwritten by requests from other clients. Since the underlying memory allocators maintain their free lists using MRU, we expect that under normal load only the most recently used blocks are reused. As a result, an attacker flooding the server with an unusually large number of parallel requests can force the allocator to reach regions of the *deep heap* which are almost never used during regular execution. To verify this intuition, this time we issued 100,000 HTTP requests using 100 (compared to the previous 1,000) concurrent connections to nginx. Our results confirmed that only three unique patterns with a given target alignment (median of 11 runs) could be found on the heap, overwriting only the 14% of the patterns sprayed on the heap by an attacker using an order of magnitude larger number of concurrent requests.

We use our heap spraying technique when disclosing password hashes in Section VII-D and heap pointers in Section VII-E.

C. Server Fingerprinting

To fingerprint a running nginx instance, an attacker needs to find one or more unique non-file-backed memory pages to deduplicate. We note that there may be other ways to

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

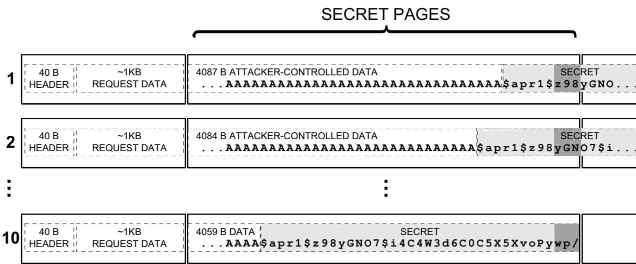

Figure 10. nginx password hash disclosure using alignment probing.

fingerprint running server programs, for instance, by sending a network request on well-known ports and looking for unique response patterns. In some cases, this is, however, not possible or not version-accurate. In addition, server fingerprinting is much more efficient with memory deduplication. In a single pass, an attacker can efficiently look for many running vulnerable programs or simply for programs with high exposure to deduplication-based attacks from a database. We stress that none of the attacks presented here exploit any software vulnerabilities.

In many running programs, it is easy for an attacker to find memory pages for fingerprinting purposes in the data section. Many of such pages are written to in a predictable way during the early stages of execution and never change again once the program reaches a steady state. Such access patterns are particularly common for server programs, which initialize most of their data structures during initialization and exhibit read-only behavior on a large fraction of them after reaching a steady state [15].

To confirm our intuition in nginx, we compared the contents of all the pages in its data segment after initialization (baseline) against the content of the same pages after running all the tests of the publicly available nginx test suite [4]. Our results showed that three out of the total eight data pages always remain identical, despite the test suite relying on a very different (and peculiar) nginx configuration compared to the standard one used in our original baseline. The attacker can abuse any of these three data pages (e.g., the data page at offset `0x2000`), or all of them for redundancy, to detect our version of nginx running next to the host browser on the same system. Once the attacker has fingerprinted the target version, she can start sending network requests from a remote client (after scanning for the server port) to craft our primitives.

D. Password Disclosure

To disclose the HTTP password hash of the `admin` user using our alignment probing primitive, an attacker needs to first control the alignment of the password hash in memory and predict neighboring data. For both conditions to happen, the attacker needs to control data which is logically allocated close to the target password hash and predict the memory allocation behavior. Both constraints are easy to satisfy in network servers.

To satisfy the first constraint, an attacker can rely on the input password provided in the HTTP request as control data. The intuition is that the target password hash is generally allocated close to the input password for authentication purposes. We confirmed this intuition in nginx (which stores the target password hash right next to the input password), but also in other common network servers such as vsftpd, proftpd, pure-ftpd, and sshd. To satisfy the second constraint, we rely on our heap spraying technique discussed in Section VII-B.

In nginx, the target password hash is allocated in the request pool right after the input password and with no alignment restrictions. In particular, on a typical (and short) HTTP authentication request for the `admin` user with the last (`Authorization`) header including the input password (such as the one issued by `wget --user=admin --password=PA$$W0RD`), nginx allocates only a single data block in the request pool. The data block consecutively stores the 40 byte pool header, around 1 KB worth of request-dependent data objects, the input password, and the target password hash. The input password is base64-encoded by the client and stored in decoded form in memory by nginx. The target password hash is by default stored in memory by nginx as follows: `$apr1$S$H`, where `apr1` is the format (MD5), `S` is the 8 byte salt, and `H` is the 22 byte base64-encoded password hash value.

To craft a alignment probing primitive, an attacker can arbitrarily increase the size of the input password (up to 4 KB) one byte at the time (even invalid base64 strings are tolerated by nginx). This would progressively shift the target password hash in memory, allowing the attacker to control its alignment and mount a deduplication-based disclosure attack. As a result of input password decoding and some data redundancy, however, a given target password hash can only be shifted at the 3 byte granularity by increasing the input size. Nevertheless, this is sufficient for an attacker to incrementally disclose three bytes of the password hash at the time. Figure 10 outlines the different stages of the attack.

To start off the attack, an attacker can send a HTTP request with a known (decoded) password pattern of 4,087 bytes. If this pattern happens to be allocated at the page boundary, then the remaining nine bytes of the page will be filled with the `$apr1$` string followed by the first three bytes of the salt within the same data block. Once the crafted page-aligned pattern is in memory, the attacker can use memory deduplication to disclose the first three bytes of the salt in a single pass. Given that the target password hash is encoded using 6 bit base64 symbols, this requires crafting 2^{18} probe pages. The attacker can then proceed to incrementally disclose the other bytes of the salt first and the password hash then, by gradually reducing the input

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

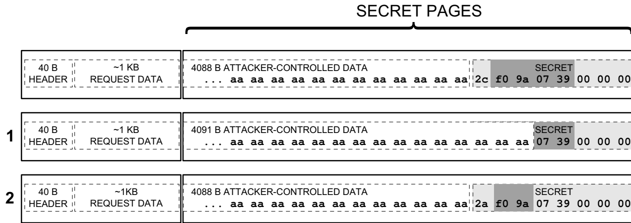

Figure 11. nginx heap disclosure using partial reuse.

password size in the HTTP request and shifting the target password hash towards lower addresses three bytes at a time.

There are three issues with this naive version of the attack: (i) the target pattern is not necessarily page-aligned, (ii) the target pattern may be overwritten by requests from other clients, and (iii) 2^{18} probe pages require 1 GB of memory without redundancy, which is large and prone to noise. To address all these issues, we rely on our heap spraying technique. Instead of issuing one request, we issue 100,000 request with our target alignment over 1,000 open connections. This allows us to reach the deep heap with our desired alignment, addressing (i) and (ii). Furthermore, thanks to the abundant redundancy when spraying the heap, the attacker can easily find many page-aligned patterns with all the three possible target password hash alignments. This enables a resource-constrained attacker to disclose two (rather than three) bytes of the password hash at the time, reducing the required memory to only 16 MB in exchange for extra deduplication passes (15 instead of 10).

E. Heap Disclosure

To leak a heap pointer (randomized using 64 bit ASLR), the alignment probing primitive used earlier is insufficient. Given that pointers are always stored in aligned objects within a data block, the attacker would be, in principle, left with guessing eight bytes at the time. In practice, Windows ASLR only uses 24 bits of entropy for the base of the heap, resulting in 36 bits of uncertainty in the lowest five bytes of arbitrary heap pointers. This is still problematic for our alignment probing primitive.

To lower the entropy, however, the attacker can deploy our partial reuse primitive by exploiting predictable memory reuse patterns. Our primitive further requires the attacker to control the alignment of a target heap pointer and some known pattern in memory. All these requirements are easy to satisfy when abusing nginx' pool allocator. To exemplify our attack, we consider the same HTTP authentication request as in our password hash disclosure attack, but we assume a (randomly crafted) invalid user in the request.

When an invalid user is specified, nginx refuses to load the target password hash into memory right after the provided input password (as done normally) and logs an error. The error logging function (`ngx_http_write_filter`), however, immediately allocates a 8 byte-aligned buffer object (`ngx_buf_t`) of 52 bytes and an unaligned log buffer in the request pool. Since the allocation behavior is deterministic, the attacker can control memory reuse patterns and partially overwrite pointers inside the buffer object to lower their entropy and incrementally disclose heap addresses. We use the first (pointer) field (`pos`) in the `ngx_buf_t` buffer object as a reference to demonstrate the attack, whose stages are outlined in Figure 11.

To start off the attack, the attacker can specify an input password with a known (decoded) pattern of 4,088 bytes. By spraying the heap, many instances of this pattern will be allocated at the page boundary, with the remaining eight bytes of the page filled with the `pos` pointer. The attacker can then send a second request of 4,091 bytes, which, in all the page-aligned pattern instances, will reuse (and override) the lowest three bytes of the old `pos` pointer data, while forcing the pool allocator to align the new `pos` pointer to eight bytes after the page boundary. This strategy leaves only 16 bits of uncertainty left in the old pointer (the first byte and the last three bytes are now known), sufficiently lowering the entropy to disclose two pointer bytes in a single memory deduplication pass.

In a second stage, the attacker can repeat the first step above to disclose the remaining lowest three bytes (the rest are now known). To lower the entropy, the attacker can rely on the fact that the `pos` pointer is always pointed into the beginning of the log buffer, i.e., exactly 52 bytes away. In other words, a `pos` pointer allocated right before the page boundary will always contain the value `0x30` in its lowest 12 bits. This leaves only 12 bits of uncertainty left in the target pointer, which the attacker can easily disclose in a second memory deduplication pass. In total, to disclose a heap pointer we require 256 MB of memory without redundancy and two deduplication passes.

F. Dealing with Noise

We fingerprinted nginx as described using three unique data pages, providing us with a redundancy factor of three to battle noise. The password hash and heap disclosure attacks described above, however, have no redundancy.

To add redundancy to our two attacks, we rely on the attacker's ability to control the content of the request and create different target patterns in memory. To this end, we can simply issue our requests using three different request types (e.g., using different input passwords) in a round-robin fashion. As discussed in Section VII-B, on average, 19 pattern pages remain in the deep heap with the desired target alignment in a steady state. Given three different request types, on average, we still obtain 6.3 memory pages in the deep heap, each with the desired target alignment but with a different attacker-controlled pattern. On the JavaScript side, the attacker can now use this redundancy to create additional pages (that target different patterns) to increase the reliability of the attacks.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

Attack	Memory	Dedup passes	Time
Fingerprinting	12 KB	1	15 Minutes
Password disclosure	48 MB	15	225 Minutes
Heap disclosure	768 MB	2	30 Minutes

Table II
TIME AND MEMORY REQUIREMENTS FOR OUR DEDUPLICATION-BASED ATTACKS AGAINST NGINX.

Website Diversity	Full Deduplication	Zero Pages Only
1	0.13	0.12
2	0.13	0.11
4	0.14	0.13
8	0.14	0.12

Table III
FULL DEDUPLICATION RATE VERSUS DEDUPLICATION RATE OF ZERO PAGES ALONE UNDER DIFFERENT SETTINGS IN MICROSOFT EDGE.

G. Time and Memory Requirements

Table II summarizes the time and memory requirements for our three deduplication-based attacks against nginx. The reported numbers assume a redundancy factor of three to deal with noise.

VIII. MITIGATION

The main motivation behind memory deduplication is to eliminate memory pages with similar content and use physical memory more efficiently. To maximize the deduplication rate, a memory deduplication system seeks to identify candidates with *any* possible content. The implicit assumption here is that many different page contents contribute to the deduplication rate. However, this property also allowed us to craft the powerful attack primitives detailed in the paper.

We now show this assumption is overly conservative in the practical cases of interest. More specifically, we show that only deduplicating *zero pages* is sufficient to achieve a nearly optimal deduplication rate, while completely eliminating the ability to program memory deduplication and perform dangerous computations. Table III compares the achievable deduplication rate of full deduplication with that of zero page deduplication in Microsoft Edge, measured in percentage of saved memory. In each experiment, we opened eight tabs visiting the most popular websites[4]. We then changed the number of websites across tabs to emulate the user's behavior and measure its impact on the deduplication rate. We call this metric "Website Diversity". For example, with diversity of eight, each tab opens a different website, and with diversity of one, each tab opens the same website. According to our measurements, deduplicating zero pages alone can retain between 84% and 93% of the deduplication rate of full deduplication. We hence recommend deduplicating zero pages alone for sensitive, network-facing applications such as browsers. In highly security-sensitive environments, full memory deduplication is generally not advisable.

IX. RELATED WORK

We discuss previous work on side channels over shared caches (Section IX-A) and deduplication (Section IX-B). We then look at the Rowhammer vulnerability (Section IX-C) we used in our end-to-end attack on Microsoft Edge.

[4]https://en.wikipedia.org/wiki/List_of_most_popular_websites

A. Side Channels over Shared Caches

Recently accessed memory locations remain in the last-level cache (LLC) shared across different cores. Accessing cached locations is considerably faster than loading them directly from memory. This timing difference has been abused to create a side channel and disclose sensitive information.

The `FLUSH+RELOAD` attack [40] leaks data from a sensitive process, such as one using cryptographic primitives, by exploiting the timing differences when accessing cached data. Irazoqui et al. [21] improve this attack, retrieving cryptographic keys in the cloud with a combination of `FLUSH+RELOAD` and a memory deduplication side channel. Using a similar attack, Zhang et al. [43] leak sensitive data to hijack user accounts and break SAML single sign-on.

The "`RELOAD`" part of the `FLUSH+RELOAD` attack assumes the attacker has access to victims' code pages either via the shared page cache or some form of memory deduplication. The `PRIME+PROBE` attack [30], [24] lifts this requirement by only relying on cache misses from the attacker's process to infer the behavior of the victim's process when processing secret data.

Oren et al. [29] use the `PRIME+PROBE` attack in a sandboxed browser tab to leak sensitive information (e.g., key presses) from a user's browser. By performing three types of `PRIME+PROBE` attacks on the CPU caches and the TLB, Hund et al. [19] map the entire address space of a running Windows kernel, breaking kernel-level ASLR.

To perform `PRIME+PROBE`, the attacker needs the mapping of memory locations to cache sets. This mapping is complex and difficult to reverse engineer in modern Intel processors [19]. Maurice et al. [26] use performance counters to simplify the reverse engineering process. As discussed in Section VI-B, we instead rely on the behavior of Windows' page allocator to quickly construct the cache eviction sets for our Rowhammer exploit. To the best of our knowledge, this is the first example of an attack using a side channel other than timing to construct such eviction sets in a sandboxed browser.

In response to numerous cache side-channel attacks, Kim et al. [22] propose a low-overhead cache isolation technique to avoid cross-talk over shared caches. By dynamically switching between diversified versions of a program, Crane et al. [12] change the mapping of program locations to cache sets, making it difficult to perform cache attacks. These techniques, however, have not (yet) become mainstream.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

B. *Side Channels over Deduplication*

Side channels over data deduplication systems can be created over stable storage or main memory.

1) Stable storage: File-based storage deduplication has been previously shown to provide a side channel to leak information on existing files and their content [18], [28]. The first instance warning users about this issue is a Microsoft Knowledge Base article that mentions a malicious user can use the deduplication side channel to leak secret information over shared deduplicated storage [1].

Harnik et al. [18] show that file deduplication at the provider's site can allow an attacker to fingerprint which files the provider stores and brute force their content if a major fraction of each file is already known. Mulazzani et al. [28] implement a similar attack on Dropbox, a popular cloud file storage service.

2) Main memory: There are several cross-VM attacks that rely on VMM-based memory deduplication to fingerprint operating systems [31] or applications [36], detect cryptographic libraries [20], and create covert channels for stealthy backdoors [38]. Gruss et al. [16] show it is possible to perform a similar attack in a sandboxed browser tab to detect running applications and open websites.

CAIN [8] can leak randomized code pointers of neighboring VMs using the memory deduplication side channel incorporated into VMMs. CAIN, however, needs to brute force all possible pointer values to break ASLR. Rather than relying on memory deduplication, Xu et al. [39] show that malicious VMMs can purposefully force page faults in a VM with encrypted memory to retrieve sensitive information.

All these previously published attacks rely on the assumption that programming memory deduplication only allows for a single-bit side channel per page. As we showed in this paper, by controlling the alignment/reuse of data in memory or mounting birthday attacks, memory deduplication can be programmed to leak high-entropy information much more efficiently. For example, by applying our alignment probing primitive to JIT code, we can leak code pointers with significantly lower memory requirements than a purely brute-force approach; by using our partial reuse primitive and our birthday heap spray primitive, we can leak high-entropy heap data pointers inside a server and a browser program (respectively) for the first time through a side channel.

C. *Rowhammer Timeline*

The Rowhammer bug was first publicly disclosed by Kim et al. [23] in June 2014. While the authors originally speculated on the security aspects of Rowhammer, it was not clear whether it was possible to fully exploit Rowhammer until later. Only in March 2015, Seaborn and Dullien [33] published a working Linux kernel privilege escalation exploit using Rowhammer. Their native exploit relies on the ability to spray physical memory with page-table entries, so that a single bit flip can probabilistically grant an attacker-controlled process access to memory storing its own page-table information. Once the process can manipulate its own page tables, the attacker gains arbitrary read and write capabilities over the entire physical memory of the machine. In July 2015, Gruss et al. [17] demonstrated the ability to flip bits inside the browser using Rowhammer.

In this paper, we showed that our memory deduplication primitives can provide us with derandomized pointers to code and heap. We used these pointers to craft the first reliable remote Rowhammer exploit in JavaScript.

X. Conclusions

Adding more and more functionality to operating systems leads to an ever-expanding attack surface. Even ostensibly harmless features like memory deduplication may prove to be extremely dangerous in the hands of an advanced attacker. In this paper, we have shown that deduplication-based primitives can do much more harm than merely providing a slow side channel. An attacker can use our primitives to leak password hashes, randomized code and heap pointers, and start off reliable Rowhammer attacks. We find it extremely worrying that an attacker who simply times write operations and then reads from an unrelated addresses can reliably "own" a system with all defenses up, even if the software is entirely free of bugs. Our conclusion is that we should introduce complex features in an operating system only with the greatest care (and after a thorough examination for side channels), and that full memory deduplication inside the operating system is a dangerous feature that is best turned off. In addition, we have shown that full deduplication is an overly conservative choice in the practical cases of interest and that deduplicating only zero pages can retain most of the memory-saving benefits of full deduplication while addressing its alarming security problems.

Disclosure Statement

We are currently working with our contacts at Microsoft to devise immediately deployable solutions against the security problems of memory deduplication evidenced in this paper.

Acknowledgments

We would like to thank our shepherd, Ilya Mironov, and the anonymous reviewers for their valuable comments. We would also like to thank the authors of Rowhammer.js for open-sourcing their implementation, which helped us create cache eviction sets inside Microsoft Edge. Finally, we would like to thank Matt Miller and Manuel Costa from Microsoft for the attention devoted to the issues raised in the paper and commitment to devise immediately deployable solutions. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571, by NWO through the project VICI "Dowser", and by the European Research Council through project ERC-2010-StG 259108 "Rosetta".

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Administrators implementing SIS with sensitive data should not share folders. https://support.microsoft.com/en-us/kb/969199.

[2] Cache and memory manager improvements in Windows Server 2012. http://goo.gl/SbhoFC.

[3] How to use the Kernel Samepage Merging feature. http://www.kernel.org/doc/Documentation/vm/ksm.txt.

[4] nginx test suite. https://github.com/catap/nginx-tests.

[5] ptmalloc. http://www.malloc.de/en.

[6] Reducing runtime memory in Windows 8. http://goo.gl/IP56lO.

[7] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *OLS*, 2009.

[8] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently breaking ASLR in the cloud. In *WOOT*, 2015.

[9] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA*, 2002.

[10] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[11] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *CCS*, 2015.

[12] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, 2015.

[13] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[14] Thomas Dullien. Exploitation and state machines: Programming the 'weird machine' revisited. In *Infiltrate*, 2011.

[15] Cristiano Giuffrida, Calin Iorgulescu, and Andrew S. Tanenbaum. Mutable checkpoint-restart: Automating live update for generic server programs. In *Middleware*, 2014.

[16] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed JavaScript. In *ESORICS*. 2015.

[17] Daniel Gruss, Clementine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. http://arxiv.org/abs/1507.06955, 2015.

[18] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security and Privacy Magazine*, 2010.

[19] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel-space ASLR. In *IEEE S&P*, 2013.

[20] Gorka Irazoqui, Mehmet Sinan IncI, Thomas Eisenbarth, and Berk Sunar. Know thy neighbor: Crypto library detection in the cloud. In *PETS*, 2015.

[21] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *ASIACCS*, 2015.

[22] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[23] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[24] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE S&P*, 2015.

[25] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *CCS*, 2015.

[26] Clementine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurelien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *RAID*, 2015.

[27] Matt Miller and Ken Johnson. Exploit mitigation improvements in Windows 8. In *Black Hat USA*, 2012.

[28] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 2011.

[29] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, 2015.

[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, 2006.

[31] R. Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IPCCC*, 2011.

[32] Shashank Rachamalla, Dabadatta Mishra, and Purushottam Kulkarni. Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems. In *HiPC*, 2013.

[33] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. http://goo.gl/OMJcj3.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.

[34] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *HPDC*, 2012.

[35] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE S&P*, 2013.

[36] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest OS. In *EuroSec*, 2011.

[37] Julien Vanegue. The automated exploitation grand challenge: Tales of weird machines. In *H2C2*, 2013.

[38] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. A covert channel construction in a virtualized environment. In *CCS*, 2012.

[39] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side-channels for untrusted operating systems. In *IEEE S&P*, 2015.

[40] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[41] Mark Yason. MemGC: Use-after-free exploit mitigation in Edge and IE on Windows 10. http://goo.gl/yUw0vY.

[42] Zhang Yunhai. Bypass control flow guard comprehensively. In *Black Hat USA*, 2015.

[43] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, 2014.

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 27,2020 at 21:03:34 UTC from IEEE Xplore. Restrictions apply.