# SECURITY ANALYSIS OF MEMORY TAGGING

Joe Bialek, Ken Johnson, Matt Miller – Microsoft Security Response Center (MSRC)
Tony Chen – Windows OS Security

## SUMMARY

Memory tagging[1] is a technology that can assist with the discovery of bugs in software and can also aid in the mitigation of vulnerabilities and exploits. In this paper, we explore some of the theoretical security properties of memory tagging as a vulnerability and exploit mitigation technology[2].

At a high level, Arm's Memory Tagging Extension (MTE) works as follows:

1. Each 16-byte aligned memory region has a 4-bit tag ("memory tag") associated with it.
2. Pointers have a 4-bit tag stored in reserved bits of the pointer ("address tag").
3. When pointers are followed, the address tag is compared against the memory tag associated with the memory address being accessed. If the tags do not match, an exception is thrown which will typically lead to a crash.

The key value proposition for memory tagging is summarized by the following table which looks at the impact memory tagging could have had on vulnerabilities reported to Microsoft in recent years[3]:

| **Vulnerability Class Mitigated** | **Deterministic or Probabilistic** | **% of Microsoft memory safety CVEs** | **Notes** |
|---|---|---|---|
| Heap overrun / overread (adjacent) | Deterministic | ~13% | Durable protection for adjacent heap memory accesses. |
| Use-after-free | Probabilistic | ~26% | Expected chance of success for an attacker: 6% if all tag bits are used by the heap allocator, unless chained together with an uninitialized memory or type confusion vulnerability.<br><br>This vulnerability class is challenging to holistically mitigate without technology like memory tagging. |
| Heap out-of-bounds read or write (non-adjacent) | Probabilistic | ~27% | Expected chance of success for an attacker: 6% if all tag bits are used by the heap allocator, unless chained together with an uninitialized memory or type confusion vulnerability.<br><br>This vulnerability class may be possible to partially mitigate without memory tagging. |

[1] See Arm Memory Tagging Extensions (MTE) https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf and SPARC Application Data Integrity (ADI) https://swisdev.oracle.com/_files/What-Is-ADI.html

[2] See also prior work on "Memory Tagging and how it improves C/C++ memory safety" https://arxiv.org/ftp/arxiv/papers/1802/1802.09517.pdf

[3] This data spans 2015 through 2019. See appendix for a general breakdown of the vulnerability classification data considered here.

| | | | |
|---|---|---|---|
| | | | Alternative solutions require rewriting code in a safer language such as Rust or leveraging safer C++ coding practices (such as GSL span). |

Based on this data, it seems like memory tagging has the potential to provide good value. It can provide the ability to dynamically discover vulnerabilities in software, it can offer deterministic and durable protection for 13% of the memory safety vulnerabilities reported to Microsoft in recent years, and it can provide non-deterministic protection for classes of vulnerabilities that are otherwise challenging to mitigate with low developer friction, and it helps detect bugs at lower cost. However, there are limits to the amount of non-deterministic protection that can be offered which is explained further in this document.

## IMPACT CATEGORIES

In this section, we explore the impact that memory tagging is expected to have as it relates to vulnerability class-specific primitives and exploitation primitives in general[4].

### CLASS-SPECIFIC PRIMITIVES

The first impact category to consider is related to the initial memory safety violation that is enabled by a particular class of vulnerability. These are referred to as class-specific primitives. For example, a heap buffer overrun provides an attacker with the ability to access adjacent memory (out-of-bounds). Each class of memory safety vulnerability maps to a set of initial primitive(s) that it can enable. If these initial primitives can be deterministically prevented, then a vulnerability class can be fully mitigated.

The following table captures the initial set of primitives that can be achieved through a memory safety vulnerability. It should be noted that some classes of vulnerabilities, such as type confusions, can vary in terms of the specific primitives that are achievable (e.g. adjacent vs. non-adjacent memory access). Furthermore, temporal issues such as race conditions can give rise to the initial primitives mentioned below.

| Category | Primitive | Explanation |
|---|---|---|
| **Spatial** | Adjacent memory access | The initial unsafe memory access **is always immediately adjacent** to the object being accessed.<br>This can happen when an attacker can cause the initial displacement of an unsafe memory access to refer to an immediately adjacent object. For example, through a traditional buffer overrun. |
| | Non-adjacent memory access | The initial unsafe memory access **may not be immediately adjacent** to the object being accessed.<br>This can happen when an attacker can influence the initial displacement of an unsafe memory access such that it refers to a non-adjacent object. For example, through an index-controlled array access. |

[4] For more information on the taxonomy used in this paper, see "Modeling the exploitation and mitigation of memory safety vulnerabilities". https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2012_10_Breakpoint/BreakPoint2012_Miller_Modeling_the_exploitation_and_mitigation_of_memory_safety_vulnerabilities.pdf

| | | |
|---|---|---|
| | Arbitrary memory access | The initial unsafe memory access **may refer to any address** in memory.<br>It is uncommon to observe this as the first primitive enabled by a vulnerability, but it can occur in cases where a value is read from an untrusted source and used as the base address of a memory access. |
| | Intra-object memory access | The initial unsafe memory access **is always within** the top-level object being accessed.<br>This can happen when an attacker can cause the initial displacement of an unsafe memory access to refer to another field within the top-level object. For example, through a traditional overflow where the array is a member field of a structure and other fields are adjacent to the array field. Another example is a type confusion vulnerability where an object of one type is incorrectly interpreted as an object of a different incompatible type (allowing object fields to be accessed and used incorrectly). |
| **Temporal** | Uninitialized memory use | The initial unsafe memory access is to **memory that has not been initialized**. |
| | Use after free (freed state) | The initial unsafe memory access is to **memory that has been freed but not yet reallocated**. |
| | Use after free (reallocated state) | The initial unsafe memory access is to **memory that has been freed and reallocated**, possibly as a different type. |

## EXPLOITATION PRIMITIVES

The second impact category to consider is related to the subsequent set of exploitation primitives that can be achieved by an attacker after the initial class-specific primitive is used. These primitives can be modeled in terms of transitions between different states of control that an attacker can achieve through a chained sequence of such primitives, ultimately resulting in a desired end-state violation being reached. For each exploitation primitive, there are five attributes:

1. **Memory access method**. This is the type of memory access which can be read, write, or execute.
2. **Base**. This is the base address of the memory access.
3. **Displacement**. This is the offset relative to the base address of the memory access.
4. **Extent**. This is the number of bytes that is being accessed relative to the base+displacement address.
5. **Content**. This is the content of the memory that is being read, written, or executed.

The base, displacement, extent, and content of a memory access can be in one of four states: controlled (by the attacker), fixed (a constant that the attacker does not directly control), uninitialized, or unknown (the attacker doesn't have knowledge of the state). For the sake of simplicity, this section will distill primitives down into transitions between relative (base not controlled) memory accesses and arbitrary (base is controlled) memory accesses. In cases where an attacker can fully address memory through a controlled displacement, we will treat this as an arbitrary memory access to further simplify this section.

| From primitive | To primitive | Explanation |
|---|---|---|

| | | |
|---|---|---|
| Relative Write | Relative Read | Corrupt memory used as the displacement and/or extent of a subsequent read |
| Relative Write | Arbitrary Read | Corrupt memory used as the base of a subsequent read |
| Relative Write | Arbitrary Write | Corrupt memory used as the base of a subsequent write |
| Arbitrary Write | Relative Read | Corrupt memory used as the displacement and/or extent of a subsequent read |
| Arbitrary Write | Arbitrary Read | Corrupt memory used as the base of a subsequent read |
| Relative Read | Relative Write | Read value used as the displacement and/or extent of a subsequent write |
| Relative Read | Arbitrary Read | Read value used as the base of a subsequent read |
| Relative Read | Arbitrary Write | Read value used as the base of a subsequent write |
| Relative Read | Arbitrary Execute | Read value used as the base of a subsequent execute |
| Arbitrary Read | Arbitrary Execute | Read value used as the base of a subsequent execute |

The linkage between the class-specific primitives enabled by an initial vulnerability and the initial primitive that they can enable is illustrated below:

| Class-specific primitive | Possible initial exploitation primitive |
|---|---|
| **Adjacent memory access** | Relative Write<br>Relative Read |
| **Non-adjacent memory access** | Relative Write<br>Relative Read |
| **Arbitrary memory access** | Arbitrary Read<br>Arbitrary Write |
| **Intra-object memory access** | Relative Write<br>Relative Read |
| **Uninitialized memory use/leak** | Relative Read<br>Relative Write<br>Arbitrary Read<br>Arbitrary Write |
| **Use after free (freed state)** | Relative Read<br>Relative Write<br>Arbitrary Read<br>Arbitrary Write |
| **Use after free (reallocated state)** | Relative Read<br>Relative Write<br>Arbitrary Read<br>Arbitrary Write |

## MEMORY TAGGING

This section assumes a memory tagging design and implementation with the following properties:

1. Each 16-byte aligned region of memory has 4 tag bits associated with it ("memory tag").
2. 4 reserved virtual address bits are used to associate an "address tag" with a pointer by an allocator.
3. When a memory access occurs, the address tag in the pointer must match the memory tag assigned to the memory region that is being referred to. If they do not match, an exception will be thrown which will lead to process termination[5].
4. The heap allocator is modified to initialize memory tags for allocations and set the corresponding address tag for the pointer returned during allocation.
    a. All heap allocations are 16-byte aligned and allocated in sizes that are multiples of 16.
    b. The heap allocator will guarantee that adjacent allocations always use distinct tags to ensure that adjacent memory accesses fault.
    c. Tags are otherwise randomly assigned.
5. There are no special tag states; the address tag value in a pointer must match for all memory tag values for the memory region being accessed.
    a. It should be noted that some software implementations may need to situationally disable tag checks which could affect the overall security efficacy and implementation complexity. One scenario where tag checks may need to be temporarily disabled would be places where shared memory is mapped across a trust boundary (e.g. kernel mapping user space memory) and a tag check failure cannot be tolerated. MTE supports this type of selective disablement through a processor state known as Tag Check Override (PSTATE.TCO) which disables tag checks when set to 1.
    b. Furthermore, not all memory accesses will necessarily generate a tag check. For example, the Arm architecture manual provides a list of memory accesses that are currently classified as Tag Unchecked for MTE[6].

In this design, the following were considered but are out of scope:

1. Stack and global buffers will not be tagged (e.g. they will use a zero tag) as these are rarely the targets of initial corruption in exploits and this is expected to help simplify the design and implementation of support for memory tagging.
2. Heap allocations are not retagged and are not set to a reserved "freed" tag when an allocation is freed as this is expected to have a poor cost/benefit tradeoff. Instead, heap allocations will retain their previous tag when freed.

## IMPACT ON CLASS-SPECIFIC PRIMITIVES

The following table summarizes the impact this design would have on 1st order class-specific primitives (i.e. the initial primitive) relative to the percentages of CVEs that map to each primitive.

| Category | Primitive | Impact | % of memory safety CVEs (2015-2019) |
|---|---|---|---|

[5] MTE supports both architecturally precise (synchronous) and architecturally imprecise (asynchronous) tag checks. In this analysis, we assume that tag checks are architecturally precise and therefore synchronously and deterministically report tag check failures at the point at which they occur.
[6] Section D6.8.1, "Tag Unchecked accesses". https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf

| | | | |
|---|---|---|---|
| **Spatial** | Adjacent memory access (heap) | Deterministically mitigated.<br>Adjacent out-of-bounds memory access will always fault for heap allocated memory.<br>It does not matter if the attacker discovers the tag used for adjacent allocations because they will not be able to modify the address tag bits of the initial access. | ~13% |
| | Non-adjacent memory access (heap) | Probabilistically mitigated.<br>Non-adjacent out-of-bounds memory accesses will probabilistically fail. ~6% probability of the initial memory access succeeding if all tag bits are used by the heap allocator.<br>If an attacker discovers tags used by non-adjacent allocations, they can create conditions to ensure their non-adjacent memory access deterministically succeeds.<br>Non-adjacent out-of-bounds memory accesses to non-heap memory would not be mitigated. | ~27% |
| | Arbitrary memory access | Barely mitigated.<br>Attempts to access tagged memory (e.g. heap) will probabilistically fail while attempts to access non-tagged memory (e.g. stack, globals, etc) will succeed assuming the address tag matches.<br>This effectively means that ASLR is also needed for protection. | ~2% |
| | Intra-object memory access | Not mitigated.<br>Object contents can be read from and written to since the entire object has the same tag. This is necessary so that operations such as memcpy can copy plain-old-data objects around.<br>These bugs may be useful for leaking valid memory to build reliable exploits. | No data available |
| **Temporal** | Uninitialized memory use/leak | Not mitigated / Probabilistically mitigated<br>Uninitialized memory use may be probabilistically mitigated or not mitigated at all depending on the nature of the bug.<br>Uninitialized memory leaks will be useful for leaking valid memory tags or leaking the address of memory that doesn't use | ~12% (over half are info disclosure only) |

| | | | |
|---|---|---|---|
| | | tagging (i.e. stack or global variable addresses).<br>Note: Other technologies exist to solve this class of issue (i.e. InitAll[7], Pool Zeroing). | |
| | Use after free (freed state) | Not mitigated.<br>Not expected to be generally useful to attackers, though, as they typically need to reallocate the freed memory. | ~26% |
| | Use after free (reallocated state) | Probabilistically mitigated.<br>Using previously freed memory via a dangling pointer will probabilistically fail. ~6% probability of the memory access succeeding if all tag bits are used by the heap allocator.<br>If an attacker discovers the tag assigned to reallocated memory matches the one assigned to the original allocation, they can create conditions to ensure that their use after free deterministically succeeds. | |

For the primitives that are only probabilistically mitigated, we need to consider the relevant exploitation primitives that could be leveraged in conjunction with these vulnerabilities and how attackers may be able to chain these primitives.

Before exploring the impact on exploitation primitives, it is helpful to consider strategies that an attacker may employ to exploit the initial primitives that are not fully mitigated.

## DISCOVERING TAGS ASSIGNED TO MEMORY

If the tag assigned to a memory allocation can be discovered, then an attacker may be able to reliably perform non-adjacent memory accesses, arbitrary memory accesses, and use freed memory that has been reallocated as part of their initial violation and possibly for subsequent primitives. Requirements for this attack:

1. The ability to read the tag assigned to a desired memory allocation.

2. The ability to create the conditions where an unsafe memory access uses a pointer that has an address tag that matches the memory tag assigned to the desired memory allocation.

We anticipate the following vulnerabilities will be useful:

1. Uninitialized memory vulnerabilities – Could allow an attacker to read pointers (including address tag bits). Could allow an attacker to effectively trigger a type confusion (force some code to follow an uninitialized pointer that happens to point at a valid object).

[7] https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_09_CppCon/CppCon2019%20-%20Killing%20Uninitialized%20Memory.pdf

2. Intra-object memory access – Could allow an attacker to read pointers contained within the object and then corrupt the pointers (or other data) with no chance of tag mismatch.

3. All other bug classes when applied to allocations that opt-out of memory tagging (i.e. object is on the heap, a global, or uses an allocator that doesn't tag memory). Could be useful to both leak memory tags and/or corrupt memory.

## DISCOVERING MEMORY THAT OPTS-OUT OF MEMORY TAGGING

In this paper we assume that memory tagging will only be applied to the heap. That means that if an attacker can locate the virtual address of zero tagged memory (such as image globals, the stack, or other memory allocated from outside the heap), the attacker can potentially corrupt this memory by overwriting pointers and using a known (zero) address tag.

## CHANGING THE TAG ASSIGNED TO MEMORY

If the tag assigned to a region of memory can be changed to an attacker-chosen value, then an attacker may be able to reliably perform non-adjacent memory accesses, arbitrary memory accesses, and use freed memory that has been reallocated as part of their initial violation and possibly for subsequent primitives. Requirements for this attack:

1. The ability to set the tag of a desired memory allocation to a desired value.

2. The ability to create the conditions where an unsafe memory access uses a pointer that matches the tag that has been assigned to the desired memory allocation.

There is seemingly no good reason for such functionality to exist.

## PARTIAL OVERWRITE WITH PREDICTABLE TAGS

Heap allocators that assign alternating patterns of tags when allocating blocks within a region of memory may make it easier to leverage a partial pointer overwrite to read or write other blocks within the region. This could occur if an attacker is able to modify the low bits of a pointer such that the pointer refers to an allocation that is a multiple of 2 blocks adjacent while retaining the same tag, thus ensuring the memory access succeeds. To mitigate this, non-repeating and non-predictable tags should be used.

## GUESSING TAG ASSIGNMENT

Due to the limited amount of entropy available for random tag assignment by an allocator, an attacker who can repeat their exploit multiple times (or against multiple machines) may be able to guess the tag assigned to an allocation with a reasonably high probability of success. This means the probabilistic protection offered by memory tagging for various primitives may not offer significant protection in all cases. Even if the target is eventually exploited, attacks may be easier to detect due to crash dumps being generated on failed attempts.

## SIDE CHANNELS

It is possible that side channel vulnerabilities (e.g. related to speculative execution or otherwise) will be found that impact memory tagging. The following types of side channels are considered likely and should be anticipated as part of the threat model:

1. Attackers can break ASLR and reliably locate images, stacks, heaps, and other memory regions. Note that these side channels already exist in many contexts (javascript -> browser process, user-mode -> kernel-mode).
2. Attacker can infer the memory tag value of memory without triggering a memory tagging violation.
3. Attacker can infer the address tag value of a pointer without triggering a memory tagging violation.

Side channel #1 does not break memory tagging against 1st order primitives.

Side channel #2 and #3 could eliminate the probabilistic failure paths for non-adjacent out-of-bounds access, use-after-free, etc. With these side channels in place, the only hard guarantee memory tagging would be able to make is that adjacent memory accesses will fail (where the heap ensures that adjacent allocations have different tags). For example, browser applications would likely fall into this category given the existence of intra-process side channels that can be used to read memory within a browser renderer process[8]. As such, applications like browsers will not be able to assume that they will benefit from the probabilistic protections offered by memory tagging.

## IMPACT ON EXPLOITATION PRIMITIVES

If we assume that the initial class-specific primitive succeeds, either because the attacker succeeded in their 6% chance (assuming all tag bits are used by the heap allocator), or due to some other technique such as an information disclosure that allowed them to reliably use the primitive, then we need to consider the exploitation primitives an attacker could leverage to achieve more control over the target application.

### THE RACE TO NTDLL

If attackers can get a working class-specific primitive that bypasses memory tagging, they will build their exploit such that it has the minimum number of potential memory tagging failures. The full scope of this topic has not been explored and will likely be researched by attackers for many years. One solid minimum bar that we can assume will happen is “The Race to NTDLL”.

Images (including global variables) are considered out of scope for memory tagging in this paper. NTDLL contains global variables that track things such as:

1. All heap global variables, from which all heap structures can be read, from which all heap allocations and their tags can be read
2. The base address of every other image in the process

If an attacker can locate NTDLL and use an arbitrary read primitive to read its global variables, we can assume that the attacker can read the entire address space of the process with correctly matching tags. We know that attackers will write exploits with the minimum number of “blind accesses” (accesses where there could be a tag mismatch) as possible, instead preferring memory accesses where the tag is known and the access is guaranteed to succeed.

Take the following structure as an example of something that could exist in any binary:

[8] https://chromium.googlesource.com/chromium/src/+/master/docs/security/side-channel-threat-model.md

```
struct
{
    PVOID ArbitraryPtr;      // Attacker can cause reads and writes through this pointer
                             // Arbitrary read/write if it can be corrupted.

    CHAR LocalArray[10];     // Local array, attacker can trigger reads from this array.
    SIZE_T LocalArrayIndex;  // Index to read from the local array. Not bounds checked
                             // at runtime since the code knows it will never exceed 9.
                             // This assumption is not true if memory corruption happens.

    PVOID NtdllPtr;          // Pointer to some code in ntdll
} MyStruct;
```

If an attacker has the ability to corrupt this structure (for example, by using a first-order relative write primitive), they can:

- Corrupt the LocalArrayIndex, then use another API to interact with this object and read beyond the bounds of LocalArray. This out-of-bounds relative read allows them to leak NtdllPtr. The attacker now knows the address of NTDLL.
- Corrupt ArbitraryPtr with the address of some offset in to NTDLL. This allows them to read NTDLL global variables and leak the address to the heap tracking structures.
- Corrupt ArbitraryPtr with the address of the heap tracking structures and leak the addresses of every heap allocation and corresponding tag.

This quickly devolves to an arbitrary read-write primitive. The only potential failure-case in this example is the first access to this structure. If the attacker blindly used a relative-write to corrupt this structure, they have a ~6% chance of success. If they had some sort of information disclosure that allowed them to reliably corrupt this structure they have a 100% chance of success. All further accesses are safe. The accesses safety is described below:

1. Relative read in LocalArray: Intra-object out-of-bounds which is not protected by memory tagging.
2. Read/Write Through ArbitraryPtr to NTDLL Globals: Image globals are not protected by memory tagging.
3. Read/Write Through ArbitraryPtr to heap structures: Pointers to the heap structures were leaked (including tag bits) so ArbitraryPtr will have the correct tag bits in the pointer.

### RACE TO NTDLL TAKEAWAYS

This exercise is meant to highlight:

1. Memory tagging has the highest chance of protecting the initial primitive.
2. Memory tagging should be assumed to be defeated if the initial primitive succeeds (probabilistically or because additional vulnerabilities like uninitialized memory leaks were used to make it reliable).
3. There may be scenarios where multiple blind accesses are needed. It is advisable to treat those scenarios as a bonus, but to not depend on them when making a value decision on memory tagging.

## IMPACT ON N$^{TH}$ ORDER PRIMITIVES

The below table generalizes and summarizes the impact that memory tagging has on primitives after the 1$^{st}$ order primitive. Note that these primitives are being analyzed in isolation, but the preconditions of multiple primitives

can likely be combined. In other words, just because two primitives each have two preconditions does not necessarily mean that there are 4 blind memory accesses required.

| Initial primitive | From primitive | To primitive | Analysis |
|---|---|---|---|
| Non-adjacent memory access | Relative Write | Relative Read | Examples<br>`// write "controlled" to &corrupted`<br>`wbuf[offset_to_corrupted] = controlled;`<br>`// relative read using corrupted index`<br>`rvalue = rbuf[corrupted];`<br><br>Analysis<br>1. The memory tag assigned to &corrupted must match the address tag of wbuf, otherwise a fault will occur.<br>2. The memory tag assigned to &rbuf[corrupted] must match the address tag assigned to rbuf, otherwise a fault will occur. |
| | Relative Write | Arbitrary Read | Examples<br>`// write "controlled" to &corrupted_rptr`<br>`wbuf[offset_to_corrupted_rptr] = controlled;`<br>`// arbitrary read using corrupted pointer`<br>`read_value = *corrupted_rptr;`<br><br>Analysis<br>1. The memory tag assigned to &corrupted_rptr must match the address tag of wbuf, otherwise a fault will occur.<br>2. The memory tag assigned to *corrupted_rptr must match the address tag assigned to corrupted_rptr, otherwise a fault will occur. |
| | Relative Write | Arbitrary Write | Examples<br>`// write "controlled" to &corrupted_wptr`<br>`wbuf[offset_to_corrupted_wptr] = controlled;`<br>`// arbitrary write using corrupted pointer`<br>`*corrupted_wptr = write_value;`<br><br>Analysis<br>1. The memory tag assigned to &corrupted_wptr must match the address tag of wbuf, otherwise a fault will occur.<br>2. The memory tag assigned to *corrupted_wptr must match the address tag assigned to corrupted_wptr, otherwise a fault will occur. |
| | Relative Write | Arbitrary Execute | Examples<br>`// write "controlled" to &corrupted_fptr`<br>`wbuf[offset_to_corrupted_fptr] = controlled;`<br>`// arbitrary execute using corrupted function pointer`<br>`(*corrupted_fptr)(...);`<br><br>Analysis<br>1. The memory tag assigned to &wbuf[offset_to_corrupted_fptr] must match the address tag for wbuf, otherwise a fault will occur. |
| | Relative Read | Relative Write | Examples<br>`// read "controlled" from offset in memory` |

| | | | |
|---|---|---|---|
| | | | controlled = rbuf[offset_to_controlled];<br>// relative write using controlled as displacement<br>wbuf[controlled] = write_value;<br><br>Analysis<br>1. The memory tag assigned to &controlled must match the address tag for rbuf, otherwise a fault will occur.<br>2. The memory tag assigned to &wbuf[controlled] must match the address tag for wbuf, otherwise a fault will occur. |
| | Relative Read | Arbitrary Write | Examples<br>// read "controlled_wptr" from offset in memory<br>controlled_wptr = rbuf[offset_to_controlled_wptr];<br>// arbitrary write using controlled pointer<br>*controlled_wptr = write_value;<br><br>Analysis<br>1. The memory tag assigned to &rbuf[offset_to_controlled_wptr] must match the address tag for rbuf, otherwise a fault will occur.<br>2. The memory tag assigned to *controller_wptr must match the address tag for controller_wptr, otherwise a fault will occur. |
| | Relative Read | Arbitrary Read | Examples<br>// read "controlled_rptr" from offset in memory<br>controlled_rptr = rbuf[offset_to_controlled_rptr];<br>// arbitrary read using controlled pointer<br>value = *controlled_rptr;<br><br>Analysis<br>1. The memory tag assigned to &rbuf[offset_to_controlled_rptr] must match the address tag for rbuf, otherwise a fault will occur.<br>2. The memory tag assigned to *controlled_rptr must match the address tag for controlled_rptr, otherwise a fault will occur. |
| | Relative Read | Arbitrary Execute | Examples<br>// read "controlled_fptr" from offset in memory<br>controlled_fptr = rbuf[offset_to_controlled_fptr];<br>// arbitrary execute using controlled function pointer<br>(*controlled_fptr)(...);<br><br>Analysis<br>1. The memory tag assigned to &rbuf[offset_to_controlled_fptr] must match the address tag for rbuf, otherwise a fault will occur. |
| Uninitialized use | Relative/Arbitrary Read/Write | Relative/Arbitrary Read/Write/Execute | See non-adjacent memory access section. The core concept is the same with the key difference being some pointer being followed is uninitialized and the attacker must be able to get it set to a valid pointer with the correct tag.<br>Example (From Relative Write to Relative Read)<br>// relative write to an uninitialized write ptr<br>uninitialized_wptr[offset_to_controlled] = value;<br>// relative read using a controlled index<br>read_value = rbuf[controlled]; |

| | | | |
|---|---|---|---|
| | | | Analysis<br>1. Attacker needs to be able to initialize uninitialized_wptr to a desired address tag and it must match the memory tag for &controlled, otherwise a fault will occur.<br>2. The memory tag assigned to &rbuf[controlled] must match the address tag assigned to rbuf, otherwise a fault will occur. |
| Use after free (reallocated) | Relative Write | Relative Read | Examples<br>`// relative write by writing to a field of a dangling ptr`<br>`dangling->offset_to_controlled = write_value;`<br>`// relative read using a controlled displacement`<br>`read_value = rbuf[controlled];`<br><br>Analysis<br>1. Attacker must be able to place &controlled such that &dangling->offset_to_controlled overlaps with it, and that the address tag bits match the memory tag of &controlled.<br>2. The memory tag assigned to &rbuf[controlled] must match the address tag of rbuf, otherwise a fault will occur. |
| | Relative Write | Arbitrary Read | Examples<br>`// arbitrary read by writing to a field of a dangling ptr`<br>`dangling->offset_to_controlled_rptr = write_value;`<br>`// arbitrary read using controlled pointer`<br>`read_value = *controlled_rptr;`<br><br>Analysis<br>1. Attacker must be able to place &controlled_rptr such that &dangling->offset_to_controlled_rptr overlaps with it, and that the address tag bits of dangling must match the memory tag of &controlled_rptr.<br>2. The memory tag assigned to *controlled_rptr must match the address tag of controlled_rptr, otherwise a fault will occur. |
| | Relative Write | Arbitrary Write | Examples<br>`// arbitrary write by writing to a field of a dangling ptr`<br>`dangling->offset_to_controlled_wptr = write_value;`<br>`// arbitrary write using controlled pointer`<br>`*controlled_wptr = write_value;`<br><br>Analysis<br>1. Attacker must be able to place &controlled_wptr such that &dangling->offset_to_controlled_wptr overlaps with it, and that the address tag bits of dangling must match the memory tag of &controlled_wptr.<br>2. The memory tag assigned to *controlled_wptr must match the address tag of controlled_wptr, otherwise a fault will occur. |
| | Arbitrary Write | Relative Read | Examples<br>`// relative read by writing to an arbitrary address`<br>`*dangling->ptr_to_controlled_offset = write_value;`<br>`// relative read using controlled offset`<br>`read_value = rbuf[controlled_offset];`<br><br>Analysis |

| | | | |
|---|---|---|---|
| | | | 1. Attacker must be able to place &controlled_offset such that &dangling->ptr_to_controlled_offset overlaps with it, and that the address tag bits match the memory tag of &rbuf.<br>2. The memory tag assigned to &rbuf[controlled_offset] must match the address tag of rbuf, otherwise a fault will occur. |
| | Arbitrary Write | Arbitrary Read | Examples<br>`// arbitrary read by writing to an arbitrary address`<br>`*dangling->ptr_to_controlled_ptr = write_value;`<br>`// arbitrary read using controlled ptr`<br>`read_value = *controlled_rptr;`<br><br>Analysis<br>1. Attacker must be able to place &controlled_rptr such that &dangling->ptr_to_controlled_ptr overlaps with it, and that the address tag bits match the memory tag of *controlled_rptr.<br>2. The memory tag assigned to *controlled_rptr must match the address tag of controlled_rptr, otherwise a fault will occur. |
| | Relative Read | Relative Write | Examples<br>`// read "controlled" from offset in memory`<br>`controlled = dangling->offset_to_controlled;`<br>`// relative write using "controlled"`<br>`wbuf[controlled] = write_value;`<br><br>Analysis<br>1. Attacker must be able to place &controlled such that &dangling->offset_to_controlled overlaps with it, and that the address tag bits match the memory tag of &controlled.<br>2. The memory tag assigned to &wbuf[controlled] must match the address tag of wbuf, otherwise a fault will occur. |
| | Relative Read | Arbitrary Read | Examples<br>`// read "controlled_rptr" from offset in memory`<br>`controlled_rptr = dangling->offset_to_controlled_ptr;`<br>`// arbitrary read using "controlled_rptr"`<br>`read_value = *controlled_rptr;`<br><br>Analysis<br>1. Attacker must be able to place controlled data such that &dangling->offset_to_controlled_ptr overlaps with it, and that the address tag must match the memory tag for &controlled_ptr.<br>2. The memory tag assigned to *controlled_rptr must match the address tag of controlled_rptr, otherwise a fault will occur. |
| | Relative Read | Arbitrary Write | Examples<br>`// read "controlled_wptr" from offset in memory`<br>`controlled_wptr = dangling->offset_to_controlled_ptr;`<br>`// arbitrary write using "controlled_wptr"`<br>`*controlled_wptr = write_value;`<br><br>Analysis<br>1. Attacker must be able to place controlled data such that &dangling->offset_to_controlled_ptr overlaps with it, and that the address tag must match the memory tag for &controlled_wptr. |

| | | | |
|---|---|---|---|
| | | | 2. The memory tag assigned to *controlled_wptr must match the address tag of controlled_wptr, otherwise a fault will occur. |
| | Relative Read | Arbitrary Execute | Examples<br>`// read "controlled_fptr" from offset in memory`<br>`controlled_fptr = dangling->offset_to_controlled_ptr;`<br>`// arbitrary execute using "controlled_fptr"`<br>`(*controlled_fptr)(...);`<br><br>Analysis<br>1. Attacker must be able to place controlled data such that &dangling->offset_to_controlled_ptr overlaps with it, and that the address tag must match the memory tag of &controlled_fptr. |
| | Arbitrary Read | Arbitrary Execute | Examples<br>`// read "controlled_fptr" from pointer in memory`<br>`controlled_fptr = *dangling->ptr_to_controlled_fptr;`<br>`// arbitrary execute using "controlled_fptr"`<br>`(*controlled_fptr)(...);`<br><br>Analysis<br>1. Attacker must be able to place controlled data such that &dangling->ptr_to_controlled_fptr overlaps with it, and that the address tag must match the memory tag of &controlled_fptr.<br>2. The memory tag assigned to *ptr_to_controlled_ptr must match the address tag of ptr_to_controlled_ptr, otherwise a fault will occur. |

## CONCLUSIONS

We believe the memory tagging implementation considered in this paper has the potential to provide good value both as a technology for discovering vulnerabilities and as a mitigation for vulnerabilities (with some limitations):

- Memory tagging can provide the ability to more effectively discover many different types of memory safety vulnerabilities at scale.
- Memory tagging can durably mitigate a common class of memory safety vulnerabilities (adjacent out-of-bounds heap memory read/write) which has accounted for ~13% of the memory safety vulnerabilities observed over the period analyzed.
- Memory tagging can probabilistically mitigate most of the remaining classes of memory safety vulnerabilities (use after free, non-adjacent out-of-bounds heap memory read/write) which have accounted for ~53% of the memory safety vulnerabilities observed over the period analyzed.
- The probabilistic protection offered by memory tagging has very low entropy and is therefore subject to realistic brute force attacks in scenarios where an attacker is willing to accept the risk of failed exploit attempts. Furthermore, attempts to exploit large populations of vulnerable targets are likely to succeed on a non-trivial proportion of the target population.
- The probabilistic protection offered by memory tagging should not be assumed to be compounding for each memory access. In other words, if an attacker succeeds with the first tagged memory access, it should be assumed that techniques will be developed to reliably perform subsequent memory accesses with low probability of failure.
- The probabilistic protection offered by memory tagging is not likely to be effective in contexts where exploitable side channel vulnerabilities exist (such as web browsers). In these contexts, an attacker will

likely be able to leverage side channel vulnerabilities to discover the tags that have been assigned to allocations without risking an architectural fault.

## APPENDIX: DATA

Statistics related to the number of memory safety CVEs in different categories affecting Microsoft that were addressed from 2015 through 2019.

| | |
|---|---|
| **Number of CVEs** | 3353 |
| **Number of memory safety CVEs** | 2117 |
| **Number of stack corruption CVEs** | 39 |
| **Number of heap out-of-bounds read CVEs** | 385 |
| **Number of heap corruption CVEs** | 454 |
| **Number of use after free CVEs** | 557 |
| **Number of type confusion CVEs** | 294 |
| **Number of uninitialized use CVEs** | 250 |
| **Number of uninitialized use CVEs (Info Disclosure only)** | 157 |
| **Number of adjacent spatial safety CVEs** | 303 |
| **Number of adjacent spatial safety CVEs (heap)** | 278 |
| **Number of non-adjacent spatial safety CVEs** | 595 |
| **Number of non-adjacent spatial safety CVEs (heap)** | 563 |
| **Number of arbitrary pointer dereference CVEs** | 41 |