MiracleObject

haraken@ / 2023 June

Status: DRAFT

TL;DR of the proposal

MiracleObject proposes a new usage of the Lightweight UaF (Use-after-Free) Detector and ADVANCED_MEMORY_SAFETY_CHECKS().

1. Run the Lightweight UaF Detector in the wild and detect many UaFs. Actionable crash reports are NOT needed. It is sufficient to know the object type that caused the UaFs.
2. Add ADVANCED_MEMORY_SAFETY_CHECKS() to the detected object types. We call the object a MiracleObject.
3. MiracleObjects have not-yet-investigated UaFs (as detected in #1) but the advanced memory checks make the attacker's exploits on the MiracleObjects significantly challenging.

The workflow #1 and #2 can be mostly automated.

[Note: MiracleObject proposes a new usage of the Lightweight UaF Detector. To avoid confusion, this document refers to the original Lightweight UaF Detector (which is intended to detect UaFs with actionable crash reports, implemented by Sergei Glazunov) as the ***Original-LUD***, and refers to the new usage of the Lightweight UaF Detector (which is intended to detect UaFs with only the object types) as the ***Extreme-LUD*** (meaning an "extremely" Lightweight UaF Detector).]

Motivation

Mitigating UaFs on on-stack pointers

We launched MiraclePtr to Windows, Mac, Linux, ChromeOS and Android. MiraclePtr replaced 25000+ C++ raw pointers with raw_ptr<> and added security protections without introducing significant performance / memory cost. According to our survey on the browser process, ~60% of UaFs are mitigated with MiraclePtr. Not only does this protect users but also will allow us to downgrade the severity of MiraclePtr-protected UaF bugs by one notch and thus improve our engineering productivity (read this article to learn more). We are now working to extend the MiraclePtr support to iOS, LaCrOS, renderer processes and more pointers.

However, the coverage of MiraclePtr won't be 100% due to the performance overhead. For example, MiraclePtr cannot be enabled on on-stack pointers (T* as local variables, and "this" pointers). According to Sergei Glazunov 's survey, currently MiraclePtr covers 55% of UaFs on supported processes. The breakdown of the remaining 45% is:

- 42% on on-stack pointers
- 25% on third-party code (we are actively working on supporting some of them)
- 14% on container<T*> (we are actively working on supporting some of them)
- 18% on others

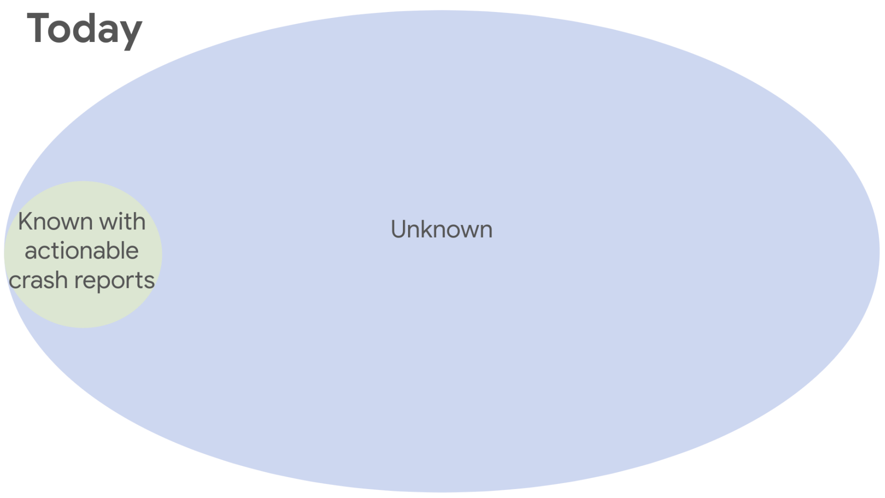
Once we support major third-party libraries (e.g., PDFium, Angle, Swiftshader) and container<T*>, **the on-stack pointers will be the majority of the remaining UaFs**. We need a solution for it.

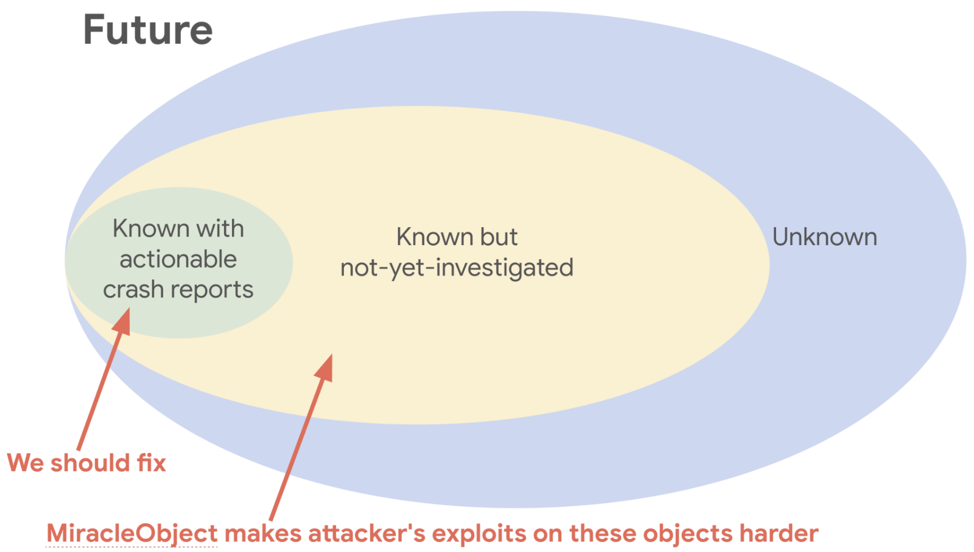
We have many unknown UaFs

As the Chrome Security team has emphasized repeatedly, UaFs we detect with GWP-ASan, Fuzzers, MiraclePtr and the VRP program are a very small subset of UaFs that exist in the code. We have three problems here:

- Problem #1: We have many unknown UaFs.
- Problem #2: The Original-LUD and MTE will help detect more UaFs, but UaFs without actionable crash reports are not useful. To make the crash reports actionable, we need to collect stack traces of the deallocation and crash. The performance overhead of collecting the actionable information limits the number of objects we can sample and decreases the detectability.
- Problem #3: Even if there is a way to detect more UaFs with actionable crash reports, it is questionable if we want to pay the engineering cost of triaging and fixing all of the detected UaFs.

We should definitely improve technologies to detect more UaFs with actionable crash reports and fix them. However, at the same time, **it is important to have a solution that makes the attacker's exploits on the unknown or known-but-not-yet-investigated UaFs more challenging**. MiracleObject provides a solution to it.




Future
Known with actionable crash reports
Known but not-yet-investigated
Unknown
We should fix
MiracleObject makes attacker's exploits on these objects harder

Goals / non-goals

From the security perspective, for a given UaF bug, we have three levels of goals (see MiraclePtr Security Properties Comparison for the detail):

- **Goal X: Reduce** real exploitability of the bug for some or all users.
- **Goal Y: Eliminate** exploitability of the bug for some or all users.
- **Goal Z: Eliminate** exploitability of the bug for **all** users with sufficient confidence that Security Sheriffs can downgrade the bug's severity.

Goals	● **Protect more not-BRP-protected pointers; specifically, achieve Goal X**

	for on-stack pointers pointing to the MiracleObjects ● **Automate the process of detecting more UaFs and annotating the detected objects as MiracleObjects** ● **Enable GWP-ASan, the Original-LUD and MTE to prioritize the MiracleObjects when sampling and detect UaFs with actionable crash reports**
Non-goals	● Achieve Goal Z for all pointers pointing to the MiracleObjects ● Replace the technologies to detect UaFs with actionable crash reports (e.g., GWP-ASan, the Original-LUD, MTE) with MiracleObject

Success metrics

- # of UaFs detected by the Extreme-LUD and # of objects annotated as MiracleObjects
- Attacker's responses (e.g., are they moving away from exploiting UaFs in Chromium?)

Proposal

#1 Detect more UaFs with the Extreme-LUD

The Original-LUD is a super lightweight version of GWP-ASan. The philosophy is: It is fine to miss UaFs as long as it can detect a lot of UaFs with moderately actionable crash reports. The goal is to minimize the performance overhead of detecting UaFs and collecting the stack traces and thus massively increase the sampling rate compared to GWP-ASan. Currently the Original-LUD can report stack traces of the deallocation and the UaF crash.

MiracleObject proposes to use the LUD in a much more lightweight manner, which we call Extreme-LUD. To achieve the goal of the MiracleObject, we do NOT need to collect the full stack traces of the deallocation and the UaF crash. **It is sufficient to understand the object type** that caused the UaF. This enables us to increase the sampling rate significantly and detect more UaFs.

The Extreme-LUD works as follows.

Sampling objects

We sample objects in PartitionAlloc::FreeNoHooks. We could add a counter per PartitionRoot, increment the counter every time an object is freed, and sample objects using the counter (e.g., using `counter % 0x1000 == 0` or doing something more advanced like GWP-ASan). Another option is to use some bits of the address of the freed object to sample objects (e.g., sample objects when `<the address of the freed object> & 0x0000000000000ff00` returns true).

We do not put the sampled objects in the thread cache.

We introduce PartitionRoot::quarantine_head. We move the sampled object to PartitionRoot::quarantine_head instead of SlotSpanMetadata::freelist_head. This prevents the object from being reused in future allocations. This should work for both directly-mapped objects and not-directly-mapped objects.

Finally we zap the object payload with 0xefefefefefefefef.

We do not collect stack traces at all at this point.

Capping the quarantine size

When the quarantine size exceeds some threshold (e.g., 2 MB), we move objects in PartitionRoot::quarantine_head to SlotSpanMetadata::freelist_head until the quarantine size goes down below e.g., half the threshold.

The threshold prevents the memory cost of the quarantined objects from growing indefinitely. This keeps the memory overhead of the Extreme-LUD within the threshold and enables us to launch it to 100% users.

Uploading a crash report

The Extreme-LUD only zaps freed objects. It is not guaranteed that it can produce a crash report at the exact point when the UaF happens. For example, you can read the zapped value (0xefefefefefefefef) without crashing and hit some unexpected crash when you use the value at a later point. Or you can overwrite the zapped value without crashing (e.g., 0xeeeeeeeeeeeeeeee). On the other hand, a virtual method call on the zapped object will crash immediately because the vtable is broken. We are fine as long as we can catch (not all but) many read-after-free + dereferences.

We upload a crash report when the crash happens on the exact zapped value (0xefefefefefefefef). At this point we collect the stack frames and upload them to the crashpad. The performance overhead of collecting the stack frames is not a problem because crashing is a rare event and also the browser is anyway crashing soon...

#2 Annotate MiracleObject

From the crash reports uploaded in the crashpad, we identify the object types that caused the UaFs [Maybe can we use Bard? 🙂].

We add ADVANCED_MEMORY_SAFETY_CHECKS() to the detected object and make it a MiracleObject.

```
class A {
    // Do not remove this macro!
    // The macro is maintained by the memory safety team.
    ADVANCED_MEMORY_SAFETY_CHECKS();
    ...;
};
```

The workflow #1 and #2 can be mostly automated. The only not-automatable part is to avoid adding ADVANCED_MEMORY_SAFETY_CHECKS() to performance-sensitive objects.

The Extreme-LUD runs on Canary / Dev / Beta / Stable on 100% users [Note: We can configure the sampling rate depending on the release channels.] and keeps detecting crashes. We should consider merging the ADVANCED_MEMORY_SAFETY_CHECKS() to previous branches when necessary.

We can manually add ADVANCED_MEMORY_SAFETY_CHECKS() in advance (i.e., before UaFs are detected by the Extreme-LUD) to objects that are likely to cause memory safety issues. Examples are RenderFrameHost, Mojo endpoint objects etc.

We might want to run a clean-up process to remove ADVANCED_MEMORY_SAFETY_CHECKS() from old objects (after confirming that the UaF concerns about the objects are gone in some way) once a year. We can worry about the clean-up process later.

#3 Mitigate UaFs on on-stack pointers

We allocate the MiracleObjects in a special partition of PartitionAlloc and guarantee the following behavior (as discussed in this email thread):

1. When the object is freed by a thread T, the object is zapped and added to T's per-thread quarantine list instead of the global free list.
2. When the thread T finishes the outermost event loop (i.e., when the stack is emptied), objects in T's quarantine list are moved to the global free list.
3. If the size of T's quarantine list exceeds some limit (e.g., 10 MB) before T finishes the outermost event loop [Note: This will be rare.], T runs a conservative stack scan and finds pointers pointing to objects in the quarantine list. Objects in the quarantine list that are not pointed to from the stack are moved to the global free list. We can use Anton Bikineev 's stack scanning (originally implemented for *Scan).

This achieves Goal X for on-stack pointers, even though it does not achieve Goal Z.

Let's see a couple of examples.

```
class A {
    ADVANCED_MEMORY_SAFETY_CHECKS();
    ...;
};

void A::SomeFunc() {
    DeleteThis();
    OtherFunc();
}

void A::DeleteThis() {
    delete this;
}

void A::OtherFunc() {
   ...;  // This does not cause Use-after-Reallocation.
}
```

`A::OtherFunc()` does not cause Use-after-Reallocation because the object is still in the quarantine and not yet returned to the global free list. The object in the quarantine is moved to the global free list when the current event loop finishes.

Next example:

```
class A {
    ADVANCED_MEMORY_SAFETY_CHECKS();
    ...;
};

void A::SomeFunc() {
    DeleteThis();
    OverflowTheQuarantine();  // Attackers may do this intentionally.
    OtherFunc();
}

void A::DeleteThis() {
    delete this;
}

void A::OverflowTheQuarantine() {
    for (int i = 0; i < 1000000; i++) {
        new A;
    }
}

void A::OtherFunc() {
   ...;  // This does not cause Use-after-Reallocation.
}
```

`A::OtherFunc()` does not cause Use-after-Reallocation because the object is still in the quarantine and not yet returned to the global free list. `A::OverflowTheQuarantine()` overflows the quarantine and triggers the stack scanning. The stack scanning is highly likely to find the "this" pointer on the machine stack or the CPU registers and keeps the object in the quarantine.

Next example:

```
class A {
    ADVANCED_MEMORY_SAFETY_CHECKS();
    ...;
};

void SomeFunc(A* a) {
    delete a;
    a->SomeFunc();  // This does not cause Use-after-Realllocation.
}
```

`a->SomeFunc()` does not cause Use-after-Reallocation because the object is still in the quarantine until the current event loop finishes.

Next example:

```
class A {
    ADVANCED_MEMORY_SAFETY_CHECKS();
    ...;
};

void SomeFunc(A* a) {
    delete a;
    OverflowTheQuarantine();  // Attackers may do this intentionally.
    a->SomeFunc();  // This does not cause Use-after-Realllocation if "a"
                    // is found on the stack.
}

void OverflowTheQuarantine() {
    for (int i = 0; i < 1000000; i++) {
        new A;
    }
}
```

`a->SomeFunc()` does not cause Use-after-Reallocation only when a is found on the stack (i.e., only when you are lucky). This is the reason Goal Z is not achieved.

#4 Enable other advanced memory safety checks

ADVANCED_MEMORY_SAFETY_CHECKS() lists other ideas to harden the memory safety of MiracleObjects:

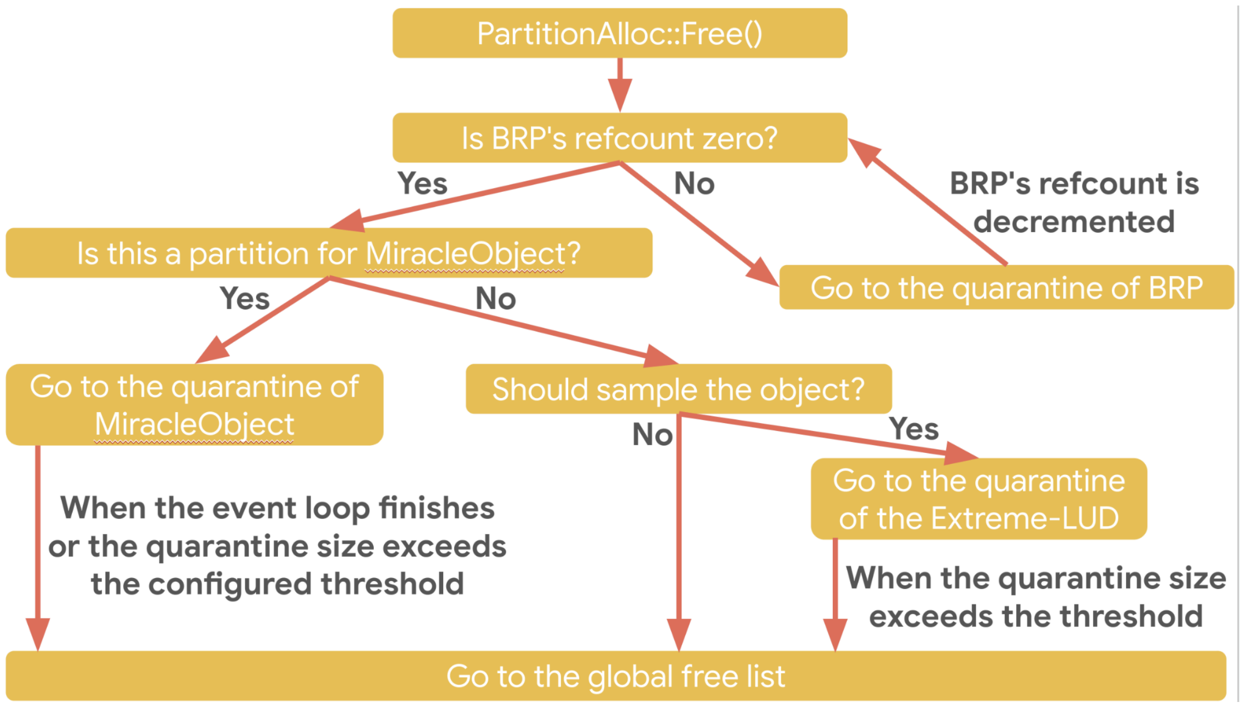
- Protect dereference-after-free
- Protect extraction-after-free
- Enable ENABLE_POINTER_SUBTRACTION_CHECK
- Enable BACKUP_REF_PTR_POISON_OOB_PTR
- go/initialize (i.e., initialize the object with memset(0) -- I'm not sure how it matters for MiracleObjects because they are zapped with 0xefefefefefefefef when they get freed.)

We can also **teach GWP-ASan and the Original-LUD to prioritize MiracleObjects when sampling**. MiracleObjects are more likely to have UaF bugs than other objects.

Discussion

The object lifetime after it's freed

The object lifetime after it's freed can be illustrated as follows:



PartitionAlloc::Free()
Is BRP's refcount zero?
Yes
No
BRP's refcount is decremented
Is this a partition for MiracleObject?
Go to the quarantine of BRP
Yes
No
Go to the quarantine of MiracleObject
Should sample the object?
No
Yes
Go to the quarantine of the Extreme-LUD
When the event loop finishes or the quarantine size exceeds the configured threshold
When the quarantine size exceeds the threshold
Go to the global free list

The performance overhead on a common path

Most objects are expected to hit the following path and go to the global free list directly:

a) Is BRP's refcount zero? -> Yes
b) Is this a partition for MiracleObject? -> No
c) Should sample the object? -> No

The performance overhead of doing these checks in PartitioAlloc::Free() will be negligible. a) already exists. b) can be done at compile time using a C++ template. c) already exists to sample objects for other purposes and we can merge the sampling logic somehow.

Three quarantines are in play

BRP, the Extreme-LUD and ADVANCED_MEMORY_SAFETY_CHECKS() have their own quarantines. These three quarantines are different things:

- The quarantine of BRP is used to delay freeing the BRP-protected objects until their reference count goes down to zero. This happens before the Extreme-LUD or ADVANCED_MEMORY_SAFETY_CHECKS() comes into play.
- The quarantine of ADVANCED_MEMORY_SAFETY_CHECKS() is used to delay freeing objects until the current event loop finishes or the quarantine size exceeds the configured threshold. This prevents objects referenced from on-stack pointers from getting Use-after-Reallocated.
- The quarantine of the Extreme-LUD is used to delay freeing randomly sampled objects for a while (until the quarantine size exceeds the configured threshold). This is used to detect UaF crash reports.
- The Extreme-LUD should be disabled on the special partition for ADVANCED_MEMORY_SAFETY_CHECKS(). This prevents an object from going to the two quarantines at the same time.

Engineering cost

Once the Original-LUD is available, it will be ~0.5 SWE quarters to implement the Extreme-LUD, ~1.5 SWE quarters to implement the protection for on-stack pointers behind ADVANCED_MEMORY_SAFETY_CHECKS(), and ~2 SWE quarters to verify the behavior and launch it.

Any comments are welcome! 😀