
GitHub
GitHub Blog
25 May, 2023

GitHub Blog

Saturday, May 27, 2023

GitHub Blog

Updates, ideas, and inspiration from GitHub to help developers build and design software.

The 2023 Open Source Program Office (OSPO) Survey is live!

Eric Sorenson

Help quantify the state of enterprise open source by taking the 2023 OSPO survey.

Rooting with root cause: finding a variant of a Project Zero bug

Man Yue Mo

In this blog, I'll look at CVE-2022-46395, a variant of CVE-2022-36449 (Project Zero issue 2327), and use it to gain arbitrary kernel code execution and root privileges from the untrusted app domain on an Android phone that uses the Arm Mali GPU. I'll also explain how root cause analysis of CVE-2022-36449 led to the discovery of CVE-2022-46395.

How to automate a Microsoft Power Platform deployment using GitHub Actions

April Edwards

Low-code enables developers and non-developers to build custom applications and solutions with less effort. In this blog, we show you how to automate your low-code deployments using GitHub Actions.

Kelsey Hightower on leadership in open source and the future of Kubernetes

Neha Batra

In this special episode of The ReadME Podcast, dedicated to GitHub's Maintainer Month, Kelsey Hightower joins hosts Martin Woodward and Neha Batra to discuss his philosophy on fostering thriving open source communities and the importance of empathy to a maintainer's success.

Announcing the public preview of GitHub Advanced Security for Azure DevOps

Walker Chabbott

GitHub Advanced Security for Azure DevOps is now available for public preview, making GitHub's same application security testing tools natively available on Azure Repos.

	Sections	

This page contains the following errors:

```
error on line 20 at column 178: redefinition of the xmlns prefix is forbidden
```

Below is a rendering of the page up to the first error.

The 2023 Open Source Program Office (OSPO) Survey is live!

The past few years have seen an increase in the number of open source program offices (OSPOs) operating in enterprises, academia, and government—notably, the Octoverse 2022 report found that over 30% of Fortune 100 companies have implemented an OSPO. These programs aim to become a center of excellence for an organization's open source activity, whether that's publicly releasing its own code, participating in upstream communities, managing its dependencies on external projects, or a combination of all of these.

Along with working on our own open source initiatives (including releasing our OSPO's policies and tools!) GitHub is partnering with the TODO Group and Linux Foundation to gather research data about OSPOs and similar open source initiatives. The project aims to provide research-backed insights into the adoption, implementation, and impact of OSPOs across different sectors and industries.

A key part of this project is a public survey that will help quantify how different organizations are addressing issues in open source, like balancing openness and control, managing external dependencies and upstream contributions, and sustaining internal and external communities. If you're involved with your organization's efforts around the use of open source software and are interested in providing your perspective, please complete

the survey! It should take less than 15 minutes and the data will be incredibly valuable to advancing our understanding of open source at scale.

Take the survey→

As a thank you, upon completion of the survey, you will receive a code for a 25% discount on any Linux Foundation e-learning training course or certification exam, as long as you register before August 30, 2023.

Rooting with root cause: finding a variant of a Project Zero bug

In this blog, I'll look at CVE-2022-46395, a variant of Project Zero issue 2327 (CVE-2022-36449) and show how it can be used to gain arbitrary kernel code execution and root privileges from the untrusted app domain on an Android phone that uses the Arm Mali GPU. I used a Pixel 6 device for testing and reported the vulnerability to Arm on November 17, 2022. It was fixed in the Arm Mali driver version r42p0, which was released publicly on January 27, 2023, and fixed in Android in the May security update. I'll go through imported memory in the Arm Mali driver, the root cause of Project Zero issue 2327, as well as exploiting a very tight race condition in CVE-2022-46395. A detailed timeline of this issue can be found here.

Imported memory in the Arm Mali driver

The Arm Mali GPU can be integrated in various devices (for example, see "Implementations" in Mali (GPU) Wikipedia entry). It has been an attractive target on Android phones and has been targeted by in-the-wild exploits multiple times.

In September 2022, Jann Horn of Google's Project Zero disclosed a number of vulnerabilities in the Arm Mali GPU driver that were collectively assigned CVE-2022-36449. One of the issues, 2327, is particularly relevant to this research.

When using the Mali GPU driver, a user app first needs to create and initialize a `kbase context` kernel object. This involves the user app opening the driver file and using the resulting file descriptor to make a series of `ioctl` calls. A `kbase_context` object is responsible for managing resources for each driver file that is opened and is unique for each file handle.

In particular, the `kbase_context` manages different types of memory that are shared between the GPU devices and user space applications. The Mali

driver provides the KBASE_IOCTL_MEM_IMPORT ioctl that allows users to share memory with the GPU via direct I/O (see, for example, the "Performing Direct I/O" section here). In this setup, the shared memory is owned and managed by the user space application. While the kernel driver is using the memory, the get_user_pages function is used to increase the refcount of the user page so that it does not get freed while the kernel is using it.

Memory imported from user space using direct I/O are represented by a kbase_va_region with the KBASE_MEM_TYPE_IMPORTED_USER_BUF kbase_memory_type.

A kbase_va_region in the Mali GPU driver represents a shared memory region between the GPU device and the host device (CPU). It contains information such as the range of the GPU addresses and the size of the region. It also contains two kbase_mem_phy_alloc pointer fields, cpu_alloc and gpu_alloc, that are responsible for keeping track of the memory pages that are mapped to the GPU. In our setting, these two fields point to the same object, so I'll only refer to them as the gpu_alloc from now on, and code snippets that use cpu_alloc should be understood to be applied to the gpu_alloc as well. In order to keep track of the pages that are currently being used by the GPU, the kbase_mem_phy_alloc contains an array, pages, that keeps track of these pages.

For KBASE_MEM_TYPE_IMPORTED_USER_BUF type of memory, the pages array in gpu_alloc is populated by using the get_user_pages function on the pages that are supplied by the user. This function increases the refcount of those pages, and then adds them to the pages array while they are in use by the GPU, and then removes them from pages and decreases their refcount once the GPU is no longer using the pages. This ensures that the pages won't be freed while the GPU is using them.

Depending on whether the user passes the KBASE_REG_SHARED_BOTH flag when the memory is imported via the KBASE_IOCTL_MEM_IMPORT ioctl, the user pages are either added to pages when the memory region is imported (when KBASE_REG_SHARED_BOTH is set), or it is only added to pages when the memory is used at a later stage.

In the case where pages are populated when the memory is imported, the pages cannot be removed until the kbase_va_region and its gpu_alloc is freed. When KBASE_REG_SHARED_BOTH is not set and the pages are populated "on demand," the memory management becomes more interesting.

Project zero issue 2327 (CVE-2022-36449)

When the pages of a KBASE_MEM_TYPE_IMPORTED_USER_BUF are not populated at import time, the memory can be used by submitting a GPU software job ("softjob") that uses the imported memory as an external resource. I can submit a GPU job via the KBASE_IOCTL_JOB_SUBMIT ioctl with the BASE_JD_REQ_EXTERNAL_RESOURCES requirement, and specify the GPU address of the shared user memory as its external resource:

```
struct base_external_resource extres = {
    .ext_resource = user_buf_addr;           //<------ GPU
address of the imported user buffer
  };
  struct base_jd_atom atom1 = {
    .atom_number = 0,
    .core_req = BASE_JD_REQ_EXTERNAL_RESOURCES,
    .nr_extres = 1,
    .extres_list = (uint64_t)&extres,
    ...
  };
  struct kbase_ioctl_job_submit js1 = {
    .addr = (uint64_t)&atom1,
    .nr_atoms = 1,
    .stride = sizeof(atom1)
  };
  ioctl(mali_fd, KBASE_IOCTL_JOB_SUBMIT, &js1);
```

When a software job requires external memory resources that are mapped as KBASE_MEM_TYPE_IMPORTED_USER_BUF, the function kbase_jd_user_buf_map is called to insert the user pages into the pages array of the gpu_alloc of the kbase_va_region via the kbase_jd_user_buf_pin_pages call:

```
static int kbase_jd_user_buf_map(struct kbase_context *kctx,
        struct kbase_va_region *reg)
{
    ...
```

```
    int err = kbase_jd_user_buf_pin_pages(kctx, reg);   //<-----
- inserts user pages
    ...
}
```

At this point, the user pages have their refcount incremented by `get_user_pages` and the physical addresses of their underlying memory are inserted into the `pages` array.

Once the software job finishes using the external resources, `kbase_jd_user_buf_unmap` is used for removing the user pages from the `pages` array and then decrementing their refcounts.

The `pages` array, however, is not the only way that these memory pages may be accessed. The kernel driver may also create memory mappings for these pages that allow them to be accessed from the GPU and the CPU, and these memory mappings should be removed before the pages are removed from the `pages` array. For example, `kbase_unmap_external_resource`, the caller of `kbase_jd_user_buf_unmap`, takes care to remove the memory mappings in the GPU by calling `kbase_mmu_teardown_pages`:

```
void kbase_unmap_external_resource(struct kbase_context *kctx,
        struct kbase_va_region *reg, struct kbase_mem_phy_alloc
*alloc)
{
    ...
    case KBASE_MEM_TYPE_IMPORTED_USER_BUF: {
        alloc->imported.user_buf.current_mapping_usage_count--;
        if (alloc->imported.user_buf.current_mapping_usage_count
== 0) {
            bool writeable = true;
            if (!kbase_is_region_invalid_or_free(reg) &&
                    reg->gpu_alloc == alloc)
                kbase_mmu_teardown_pages(
//kbdev,
                        &kctx->mmu,
                        reg->start_pfn,
                        kbase_reg_current_backed_size(reg),
                        kctx->as_nr);
            if (reg && ((reg->flags & (KBASE_REG_CPU_WR |
KBASE_REG_GPU_WR)) == 0))
                writeable = false;
            kbase_jd_user_buf_unmap(kctx, alloc, writeable);
        }
```

```
    }
    ...
}
```

It is also possible to create mappings from these userspace pages to the CPU by calling `mmap` on the Mali drivers file with an appropriate page offset. When the userspace pages are removed, these CPU mappings should be removed by calling the `kbase_mem_shrink_cpu_mapping` function to prevent them from being accessed from the `mmap'ed` user space addresses. This, however, was not done when user pages were removed from the `KBASE_MEM_TYPE_IMPORTED_USER_BUF` memory region, meaning that, once the refcounts of these user pages were reduced in `kbase_jd_user_buf_unmap`, these pages could be freed while CPU mappings to these pages created by the Mali driver still had access to them. This, in particular, meant that after these pages were freed, they could still be accessed from the user application, creating a use-after-free condition for memory pages that was easy to exploit.

Root cause analysis can sometimes be more of an art than a science and there can be many valid, but different views of what causes a bug. While at one level, it may look like it is simply a case where some cleanup logic is missing when the imported user memory is removed, the bug also highlighted an interesting deviation in how imported memory is managed in the Mali GPU driver.

In general, shared memory in the Mali GPU driver is managed via the `gpu_alloc` of the `kbase_va_region` and there are two different cases where the backing pages of a region can be freed. First, if the `gpu_alloc` and the `kbase_va_region` themselves are freed, then the backing pages of the `kbase_va_region` are also going to be freed. To prevent this from happening, when the backing pages are used by the kernel, references of the corresponding `gpu_alloc` and `kbase_va_region` are usually taken to prevent them from being freed. When a CPU mapping is created via `kbase_cpu_mmap`, a `kbase_cpu_mapping` structure is created and stored as the `vm_private_data` of the created virtual memory (vm) area. The `kbase_cpu_mapping` stores and increases the refcount of both the `kbase_va_region` and `gpu_alloc`, preventing them from being freed while the vm area is in use.

```
static int kbase_cpu_mmap(struct kbase_context *kctx,
        struct kbase_va_region *reg,
        struct vm_area_struct *vma,
        void *kaddr,
        size_t nr_pages,
        unsigned long aligned_offset,
        int free_on_close)
{
    struct kbase_cpu_mapping *map;
    int err = 0;
    map = kzalloc(sizeof(*map), GFP_KERNEL);
    ...
    vma->vm_private_data = map;
    ...
    map->region = kbase_va_region_alloc_get(kctx, reg);
    ...
    map->alloc = kbase_mem_phy_alloc_get(reg->cpu_alloc);
    ...
}
```

When the memory region is of type `KBASE_MEM_TYPE_NATIVE`, its backing pages are owned and maintained by memory region, the backing pages can also be freed by shrinking the backing store. For example, using the `KBASE_IOCTL_MEM_COMMIT` `ioctl`, would call `kbase_mem_shrink` to remove the backing pages:

```
int kbase_mem_shrink(struct kbase_context *const kctx,
        struct kbase_va_region *const reg, u64 new_pages)
{
    ...
    err = kbase_mem_shrink_gpu_mapping(kctx, reg,
            new_pages, old_pages);
    if (err >= 0) {
        /* Update all CPU mapping(s) */
        kbase_mem_shrink_cpu_mapping(kctx, reg,
                new_pages, old_pages);
        kbase_free_phy_pages_helper(reg->cpu_alloc, delta);
        ...
    }
    ...
}
```

In the above, both `kbase_mem_shrink_gpu_mapping` and `kbase_mem_shrink_cpu_mapping` are called to remove potential uses of the backing pages before they are freed. As `kbase_mem_shrink` frees the backing

pages by calling `kbase_free_phy_pages_helper`, it only makes sense to shrink a region where the backing store is owned by the GPU. Even in this case, care must be taken to remove potential references to the backing pages before freeing them. For other types of memory, references to the backing pages may exist outside of the memory region, resizing it is generally forbidden and the `pages` array is immutable throughout the lifetime of the `gpu_alloc`. In this case, the backing pages should live as long as the `gpu_alloc`.

This makes the semantics of `KBASE_MEM_TYPE_IMPORTED_USER_BUF` region interesting. While it's backing pages are owned by the user space application that creates it, as we have seen, its backing store, which is stored in the `pages` array of its `gpu_alloc`, can indeed change and backing pages can be freed while the `gpu_alloc` is still alive. Recall that if `KBASE_REG_SHARED_BOTH` is not set when the region is created, its backing store will only be set when it is used as an external resource in a GPU job, in which `kbase_jd_user_buf_pin_pages` is called to insert user pages to its backing store:

```
static int kbase_jd_user_buf_map(struct kbase_context *kctx,
        struct kbase_va_region *reg)
{
    ...
    int err = kbase_jd_user_buf_pin_pages(kctx, reg);   //<-----
- inserts user pages
    ...
}
```

The backing store then shrinks back to zero after the job has finished using the memory region by calling `kbase_jd_user_buf_unmap`. This, as we have seen, can result in cleanup logic from `kbase_mem_shrink` being omitted by mistake due to the need to reimplement complex memory management logic. However, this also means the backing store of a region can be removed without going through `kbase_mem_shrink` or freeing the region, which is unusual and may break the assumptions made in other parts of the code.

CVE-2022-46395

The idea is to look for code that accesses the backing store of a memory region and see if it implicitly assumes that the backing store is only removed when either of the followings happens:

1. When the memory region is freed, or
2. When the backing store is shrunk by the `kbase_mem_shrink` call

It turns out that the `kbase_vmap_prot` function had made these assumptions. The function `kbase_vmap_prot` is used by the driver to temporarily map the backing pages of a memory region to the kernel address space via `vmap` so that it can access them. It calls `kbase_vmap_phy_pages` to perform the mapping. To prevent the region from being freed while the `vmap` is valid, a `kbase_vmap_struct` is created for the lifetime of the mapping, which also holds a reference to the `gpu_alloc` of the `kbase_va_region`:

```
static int kbase_vmap_phy_pages(struct kbase_context *kctx,
        struct kbase_va_region *reg, u64 offset_bytes, size_t
size,
        struct kbase_vmap_struct *map)
{
    ...
    map->cpu_alloc = reg->cpu_alloc;
    ...
    map->gpu_alloc = reg->gpu_alloc;
    ...
    kbase_mem_phy_alloc_kernel_mapped(reg->cpu_alloc);
    return 0;
}
```

The refcounts of `map->cpu_alloc` and `map->gpu_alloc` are incremented in `kbase_vmap_prot` before entering this function. To prevent the backing store from being shrunk by `kbase_mem_commit`, `kbase_vmap_phy_pages` also calls `kbase_mem_phy_alloc_kernel_mapped`, which increments `kernel_mappings` in the `gpu_alloc`:

```
static inline void
kbase_mem_phy_alloc_kernel_mapped(struct kbase_mem_phy_alloc
*alloc)
{
    atomic_inc(&alloc->kernel_mappings);
}
```

This prevents `kbase_mem_commit` from shrinking the backing store of the memory region while it is mapped by `kbase_vmap_prot`, as `kbase_mem_commit` will check the `kernel_mappings` of a memory region:

```
int kbase_mem_commit(struct kbase_context *kctx, u64 gpu_addr,
u64 new_pages)
{
    ...
    if (atomic_read(&reg->cpu_alloc->kernel_mappings) > 0)
        goto out_unlock;
    ...
}
```

When `kbase_mem_shrink` is used outside of `kbase_mem_commit`, it is always used within the `jctx.lock` of the corresponding `kbase_context`. As mappings created by `kbase_vmap_prot` are only valid with this lock held, other uses of `kbase_mem_shrink` cannot free the backing pages while the mappings are in use either.

However, as we have seen, a `KBASE_MEM_TYPE_IMPORTED_USER_BUF` memory region can remove its backing store without going through `kbase_mem_shrink`. In fact, the `KBASE_IOCTL_STICKY_RESOURCE_UNMAP` can be used to trigger `kbase_unmap_external_resource` to remove its backing pages without holding the `jctx.lock` of the `kbase_context`. This means that many uses of `kbase_vmap_prot` are vulnerable to a race condition that can remove its `vmap'ed` page while the mapping is in use, causing a use-after-free in the memory pages. For example, the `KBASE_IOCTL_SOFT_EVENT_UPDATE` `ioctl` calls the `kbase_write_soft_event_status`, which uses `kbase_vmap_prot` to create a mapping, and then unmap it after the kernel finishes writing to it:

```
static int kbasep_write_soft_event_status(
        struct kbase_context *kctx, u64 evt, unsigned char
new_status)
{
    ...
    mapped_evt = kbase_vmap_prot(kctx, evt, sizeof(*mapped_evt),
                        KBASE_REG_CPU_WR, &map);
    //Race window start
    if (!mapped_evt)
        return -EFAULT;
    *mapped_evt = new_status;
```

```
    //Race window end
    kbase_vunmap(kctx, &map);
    return 0;
}
```

If the memory region that `evt` belongs to is of type `KBASE_MEM_TYPE_IMPORTED_USER_BUF`, then between `kbase_vmap_prot` and `kbase_vunmap`, the memory region can have its backing pages removed by another thread using the `KBASE_IOCTL_STICKY_RESOURCE_UNMAP ioctl`. There are other uses of `kbase_vmap_prot` in the driver, but they follow a similar usage pattern and the use in `KBASE_IOCTL_SOFT_EVENT_UPDATE` has a simpler call graph, so I'll stick to it in this research.

The problem? The race window is very, very tiny.

Winning a tight race and widening the race window

The race window in this case is very tight and consists of very few instructions, so even hitting it is hard enough, let alone trying to free and replace the backing pages inside this tiny window. In the past, I've used a technique from Exploiting race conditions on [ancient] Linux of Jann Horn to widen the race window. While the technique can certainly be used to widen the race window here, it lacks the fine control in timing that I need here to hit the small race window. Fortunately, another technique that was also developed by Jann Horn in Racing against the clock—hitting a tiny kernel race window is just what I need here.

The main idea of controlling race windows on the Linux kernel using these techniques is to interrupt a task inside the race window, causing it to pause. By controlling the timing of interrupts and the length of these pauses, the race window can be widened to allow other tasks to run within it. In the Linux kernel, there are different ways in which a task can be interrupted.

The technique in Exploiting race conditions on [ancient] Linux uses task priorities to manipulate interrupts. The idea is to pin a low priority task on a CPU, and then run another task with high priority on the same CPU during the race window. The Linux kernel scheduler will then interrupt the low

priority task to allow the high priority task to run. While this can stop the low priority task for a long time, depending on how long it takes the high priority task to run, it is difficult to control the precise timing of the interrupt.

The Linux kernel also provides APIs that allow users to schedule an interrupt at a precise time in the future. This allows more fine-grain control in the timing of the interrupts and was explored in Racing against the clock—hitting a tiny kernel race window. One such API is the `timerfd`. A `timerfd` is a file descriptor where its availability can be scheduled using the hardware timer. By using the `timerfd_settime` syscall, I can create a `timerfd`, and schedule it to be ready for use in a future time. If I have epoll instances that monitor the `timerfd`, then by the time the `timerfd` is ready, the epoll instances will be iterated through and be woken up.

```
  migrate_to_cpu(0);   //<------- pin this task to a cpu

  int tfd = timerfd_create(CLOCK_MONOTONIC, 0);   //<-----
creates timerfd
  //Adds epoll watchers
  int epfds[NR_EPFDS];
  for (int i=0; i<NR_EPFDS; i++)
    epfds[i] = epoll_create1(0);

  for (int i=0; i<NR_EPFDS; i++) {
    struct epoll_event ev = { .events = EPOLLIN };
    epoll_ctl(epfd[i], EPOLL_CTL_ADD, fd, &ev);
  }

  timerfd_settime(tfd, TFD_TIMER_ABSTIME, ...);  //<-----
schedule tfd to be available at a later time

  ioctl(mali_fd, KBASE_IOCTL_SOFT_EVENT_UPDATE,...); //<---- tfd
becomes available and interrupts this ioctl
```

In the above, I created a `timerfd`, `tfd`, using `timerfd_create`, and then added epoll watchers to it using the `epoll_ctl` syscall. After this, I schedule `tfd` to be available at a precise time in the future, and then run the `KBASE_IOCTL_SOFT_EVENT_UPDATE` `ioctl`. If the `tfd` becomes available while the `KBASE_IOCTL_SOFT_EVENT_UPDATE` is running, then it'll be interrupted and the epoll watchers of `tfd` are processed instead. By creating

a large list of epoll watchers and scheduling `tfd` so that it becomes available inside the race window of `KBASE_IOCTL_SOFT_EVENT_UPDATE`, I can widen the race window enough to free and replace the backing stores of my `KBASE_MEM_TYPE_IMPORTED_USER_BUF` memory region. Having said that, the race window is still very difficult to hit and most attempts to trigger the bug will fail. This means that I need some ways to tell whether the bug has triggered before I continue with the next step of the exploit. Recall that the race window happens between the calls `kbase_vmap_prot` and `kbase_vunmap`:

```
static int kbasep_write_soft_event_status(
        struct kbase_context *kctx, u64 evt, unsigned char
new_status)
{
    ...
    mapped_evt = kbase_vmap_prot(kctx, evt, sizeof(*mapped_evt),
                    KBASE_REG_CPU_WR, &map);
    //Race window start
    if (!mapped_evt)
        return -EFAULT;
    *mapped_evt = new_status;
    //Race window end
    kbase_vunmap(kctx, &map);
    return 0;
}
```

The call `kbase_vmap_prot` holds the `kctx->reg_lock` for almost the entire duration of the function:

```
void *kbase_vmap_prot(struct kbase_context *kctx, u64 gpu_addr,
size_t size,
              unsigned long prot_request, struct
kbase_vmap_struct *map)
{
    struct kbase_va_region *reg;
    void *addr = NULL;
    u64 offset_bytes;
    struct kbase_mem_phy_alloc *cpu_alloc;
    struct kbase_mem_phy_alloc *gpu_alloc;
    int err;
    kbase_gpu_vm_lock(kctx);         //reg_lock
    ...
out_unlock:
    kbase_gpu_vm_unlock(kctx);      //reg_lock
    return addr;
```

```
fail_vmap_phy_pages:
    kbase_gpu_vm_unlock(kctx);
    kbase_mem_phy_alloc_put(cpu_alloc);
    kbase_mem_phy_alloc_put(gpu_alloc);
    return NULL;
}
```

A common case of failure is when the interrupt happens during the `kbase_vmap_prot` function. In this case, the `kctx->reg_lock`, which is a mutex, is held. To test whether the mutex is held when the interrupt happens, I can make an `ioctl` call that requires the `kctx->reg_lock` during the interrupt from a different thread. There are many options, and I choose `KBASE_IOCTL_MEM_FREE` because this requires the `kctx->reg_lock` most of the time and can be made to return early if an invalid argument is supplied. If the `kctx->reg_lock` is held by `KBASE_IOCTL_SOFT_EVENT_UPDATE` (which calls `kbase_vmap_prot`) while the interrupt happens, then `KBASE_IOCTL_MEM_FREE` cannot proceed and would return after the `KBASE_IOCTL_SOFT_EVENT_UPDATE`. Otherwise, the `KBASE_IOCTL_MEM_FREE` `ioctl` will return first. By comparing the time when these `ioctl` calls return, I can determine whether the interrupt happened inside the `kctx->reg_lock`. Moreover, if the interrupt happens inside `kbase_vmap_prot`, then the address `evt` that I supplied to `kbasep_write_soft_event_status` would not have been written to.

So, if both of the following are true, then I know the interrupt must have happened before the race window ended, but not inside the `kbase_vmap_prot` function:

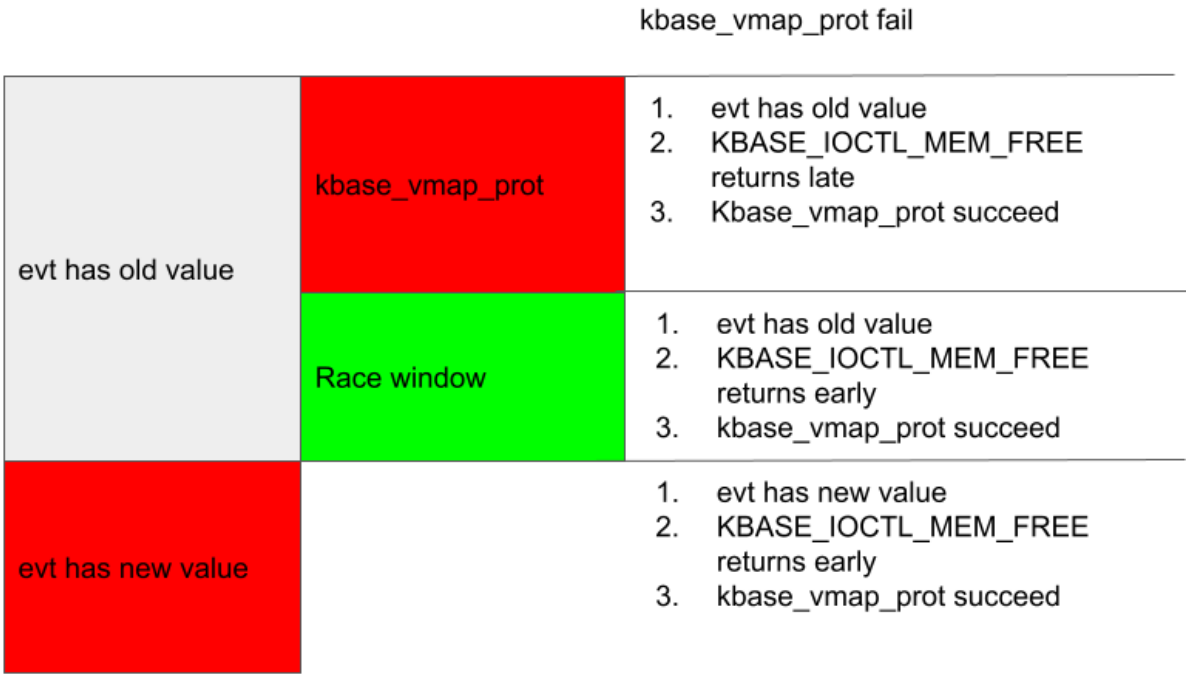
1. If an `KBASE_IOCTL_MEM_FREE` started during the interrupt in another thread returns before the `KBASE_IOCTL_SOFT_EVENT_UPDATE`
2. The address `evt` has not been written to

The above conditions, however, can still be true if the interrupt happens before `kbase_vmap_prot`. In this case, if I remove the backing pages from the `KBASE_MEM_TYPE_IMPORTED_USER_BUF` memory region, then the `kbase_vmap_prot` call would simply fail because the address `evt`, which belongs to the memory region, is no longer invalid. This then results in the `KBASE_IOCTL_SOFT_EVENT_UPDATE` returning an error.

This gives me an indicator of when the interrupt happened and whether I should proceed with the exploit or try triggering the bug again. To decide whether the race is won, I can then do the following during the interrupt:

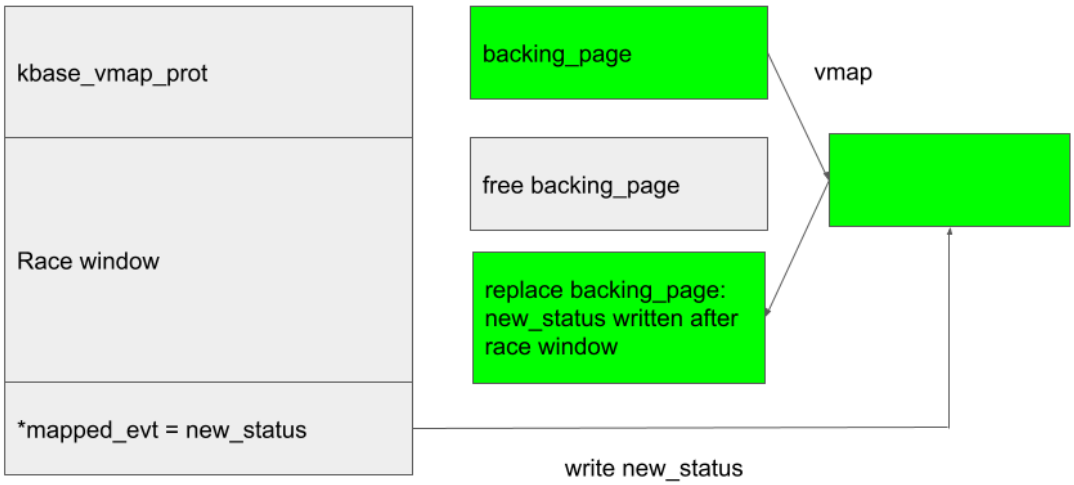
1. Check if `evt` is written to, if it is, then the interrupt happened too late and the race was lost.
2. If `evt` is not written to, then make a `KBASE_IOCTL_MEM_FREE` `ioctl`from another thread. If the `ioctl` returns and before the `KBASE_IOCTL_SOFT_EVENT_UPDATE` `ioctl` that is being interrupted, then proceed to the next step, otherwise, the interrupt happened inside `kbase_vmap_prots` and the race was lost. (interrupt happens too early).
3. Proceed to remove the backing pages of the `KBASE_MEM_TYPE_IMPORTED_USER_BUF` region that `evt` belongs to. If the `KBASE_IOCTL_SOFT_EVENT_UPDATE` `ioctl` returns an error, then the interrupt happened before the `kbase_vmap_prot` call and the race was lost. Otherwise, the race is likely won and I can proceed to the next stage in the exploit.

The following figure illustrates these conditions and their relations to the race window.



One byte to root them all

Once the race is won, I can proceed to free and then replace the backing pages of the `KBASE_MEM_TYPE_IMPORTED_USER_BUF` region. Then, after the interrupt returns, `KBASE_IOCTL_SOFT_EVENT_UPDATE` will write `new_status` to the free'd (and now replaced) backing page of the `KBASE_MEM_TYPE_IMPORTED_USER_BUF` region via the kernel address created by `vmap`.



I'd like to replace those pages with memory pages used by the kernel. The problem here is that memory pages in the Linux kernel are allocated according to their zones and migrate type, and pages do not generally get allocated to a different zone or migrate type. In our case, the backing pages of the `KBASE_MEM_TYPE_IMPORTED_USER_BUF` come from a user space application, which are generally allocated with the `GFP_HIGHUSER` or the `GFP_HIGHUSER_MOVABLE` flag. On Android, which lacks the `ZONE_HIGHMEM` zone, this translates into an allocation in the `ZONE_NORMAL` with `MIGRATE_UNMOVABLE` (for `GFP_HIGHUSER` flag), or `MIGRATE_MOVABLE` (for `GFP_HIGHUSER_MOVABLE` flag) migrate type. Memory pages used by the kernel, such as those used by the SLUB allocator for allocating kernel objects, on the other hand, are allocated in the `ZONE_NORMAL` with

MIGRATE_UNMOVABLE migration type. Many user space memory, such as those allocated via the mmap syscall, are allocated with the GFP_HIGHUSER_MOVABLE flag, making them unsuitable for my purpose. In order to replace the backing pages with kernel memory pages, I therefore need to find a way to map pages to user space that are allocated with the GFP_HIGHUSER flag.

The situation is similar to what I had with "The code that wasn't there: Reading memory on an Android device by accident." In the section "Leaking Kernel memory," I used the asynchronous I/O file system to allocate user space memory with the GFP_HIGHUSER flag. By first allocating user space memory with the asynchronous I/O file system and then importing that memory to the Mali driver to create a KBASE_MEM_TYPE_IMPORTED_USER_BUF region with that memory as backing pages, I can create a KBASE_MEM_TYPE_IMPORTED_USER_BUF memory region with backing pages in ZONE_NORMAL and the MIGRATE_UNMOVABLE migrate type, which can be reused as kernel pages.

One big problem with bugs in kbase_vmap_prot is that, in all uses of kbase_vmap_prot, there is very little control of the write value. In the case of the KBASE_IOCTL_SOFT_EVENT_UPDATE, it is only possible to write either zero or one to the chosen address:

```
static int kbasep_write_soft_event_status(
        struct kbase_context *kctx, u64 evt, unsigned char
new_status)
{
    ...
    if ((new_status != BASE_JD_SOFT_EVENT_SET) &&
        (new_status != BASE_JD_SOFT_EVENT_RESET))
        return -EINVAL;
    mapped_evt = kbase_vmap_prot(kctx, evt, sizeof(*mapped_evt),
                    KBASE_REG_CPU_WR, &map);
    ...
    *mapped_evt = new_status;
    kbase_vunmap(kctx, &map);
    return 0;
}
```

In the above, new_status, which is the value to be written to the address evt, is checked to ensure that it is either BASE_JD_SOFT_EVENT_SET, or BASE_JD_SOFT_EVENT_RESET, which are one or zero, respectively.

Even though the write primitive is rather restrictive, by replacing the backing page of my `KBASE_MEM_TYPE_IMPORTED_USER_BUF` region with page table global directories (PGD) used by the kernel, or with pages used by the SLUB allocator, I still have a fairly strong primitive.

However, since the bug is rather difficult to trigger, ideally, I'd like to be able to replace the backing page reliably and finish the exploit by triggering the bug once only. This makes replacing the pages with kernel PGD or SLUB allocator backing pages less than ideal here, so let's have a look at another option.

While most kernel objects are allocated via variants of the `kmalloc` call, which uses the SLUB allocator to allocate the object, large objects are sometimes allocated using variants of the `vmalloc` call. Unlike `kmalloc`, `vmalloc` allocates memory at the granularity of pages and it takes the page directly from the kernel page allocator. While `vmalloc` is inefficient for small locations, for allocations of objects larger than the size of a page, `vmalloc` is often considered a more optimal choice. It is also considered more secure as the allocated memory is used exclusively by the allocated object and a guard page is often inserted at the end of the memory. This means that any out-of-bounds access is likely to either hit unused memory or the guard page. In our case, however, replacing the backing page with a `vmalloc` object is just what I need. To optimize allocation, the kernel page allocator maintains a per CPU cache which it uses to keep track of pages that are recently freed on each CPU. New allocations from the same CPU are simply given the most recently freed page on that CPU from the per CPU cache. So by freeing the backing pages of a `KBASE_MEM_TYPE_IMPORTED_USER_BUF` region on a CPU, and then immediately allocating an object via `vmalloc`, the newly allocated object will reuse the backing pages of the `KBASE_MEM_TYPE_IMPORTED_USER_BUF` region. This allows me to write either zero or one to any offset in this object. A suitable object allocated by `vzalloc` (a variant of `vmalloc` that zeros out the allocated memory) is none but the `kbase_mem_phy_alloc` itself. The object is created by `kbase_alloc_create`, which can be triggered via many `ioctl` calls such as the `KBASE_IOCTL_MEM_ALLOC`:

```
static inline struct kbase_mem_phy_alloc *kbase_alloc_create(
        struct kbase_context *kctx, size_t nr_pages,
        enum kbase_memory_type type, int group_id)
```

```
{
    ...
    size_t alloc_size = sizeof(*alloc) + sizeof(*alloc->pages) *
nr_pages;
    ...
    /* Allocate based on the size to reduce internal
fragmentation of vmem */
    if (alloc_size > KBASE_MEM_PHY_ALLOC_LARGE_THRESHOLD)
        alloc = vzalloc(alloc_size);
    else
        alloc = kzalloc(alloc_size, GFP_KERNEL);
    ...
}
```

When creating a `kbase_mem_phy_alloc` object, the allocation size, `alloc_size` depends on the size of the region to be created. If `alloc_size` is larger than the `KBASE_MEM_PHY_ALLOC_LARGE_THRESHOLD`, then `vzalloc` is used for allocating the object. By making a `KBASE_IOCTL_MEM_ALLOC` `ioctl` call immediately after the `KBASE_IOCTL_STICKY_RESOURCE_UNMAP` `ioctl` call that frees the backing pages of a `KBASE_MEM_TYPE_IMPORTED_USER_BUF` memory region, I can reliably replace the backing page with a kernel page that holds a `kbase_mem_phy_alloc` object:

```
ioctl(mali_fd, KBASE_IOCTL_STICKY_RESOURCE_UNMAP, ...);  //<----
-- frees backing page
ioctl(mali_fd, KBASE_IOCTL_MEM_ALLOC, ...);              //<----
-- reclaim backing page as kbase_mem_phy_alloc
```

So, what should I rewrite in this object? There are many options, for example, rewriting the `kref` field can easily cause a refcounting problem and turn this into a UAF of a `kbase_mem_phy_alloc`, which is easy to exploit. It is, however, much simpler to just set the `gpu_mappings` field to zero:

```
struct kbase_mem_phy_alloc {
    struct kref           kref;
    atomic_t              gpu_mappings;
    atomic_t              kernel_mappings;
    size_t                nents;
    struct tagged_addr    *pages;
    ...
}
```

The Mali driver allows memory regions to share the same backing pages via the KBASE_IOCTL_MEM_ALIAS ioctl call. A memory region created by KBASE_IOCTL_MEM_ALLOC can be aliased by passing it as a parameter in the call to KBASE_IOCTL_MEM_ALIAS:

```
  union kbase_ioctl_mem_alloc alloc = ...;
  ...
  ioctl(mali_fd, KBASE_IOCTL_MEM_ALLOC, &alloc);
  void* region = mmap(NULL, ..., mali_fd, alloc.out.gpu_va);
  union kbase_ioctl_mem_alias alias = ...;
  ...
  struct base_mem_aliasing_info ai = ...;
  ai.handle.basep.handle = (uint64_t)region;
  ...
  alias.in.aliasing_info = (uint64_t)(&ai);
  ioctl(mali_fd, KBASE_IOCTL_MEM_ALIAS, &alias);
  void* alias_region = mmap(NULL, ..., mali_fd,
alias.out.gpu_va);
```

In the above, a memory region is created using KBASE_IOCTL_MEM_ALLOC, and mapped to region. This region is then passed to the KBASE_IOCTL_MEM_ALIAS call. After mapping the result to user space, both region and alias_region share the same backing pages. As both regions now share the same backing pages, region must be prevented from resizing via the KBASE_IOCTL_MEM_COMMIT ioctl, otherwise the backing pages may be freed while it is still mapped to the alias_region:

```
  union kbase_ioctl_mem_alloc alloc = ...;
  ...
  ioctl(mali_fd, KBASE_IOCTL_MEM_ALLOC, &alloc);
  void* region = mmap(NULL, ..., mali_fd, alloc.out.gpu_va);
  union kbase_ioctl_mem_alias alias = ...;
  ...
  struct base_mem_aliasing_info ai = ...;
  ai.handle.basep.handle = (uint64_t)region;
  ...
  alias.in.aliasing_info = (uint64_t)(&ai);
  ioctl(mali_fd, KBASE_IOCTL_MEM_ALIAS, &alias);
  void* alias_region = mmap(NULL, ..., mali_fd,
alias.out.gpu_va);

  struct kbase_ioctl_mem_commit commit = ...;
  commit.gpu_addr = (uint64_t)region;
```

```
  ioctl(mali_fd, KBASE_IOCTL_MEM_COMMIT, &commit);  //<----
ioctl fail as region cannot be resized
```

This is achieved using the `gpu_mappings` field in the `gpu_alloc` of a `kbase_va_region`. The `gpu_mappings` field keeps track of the number of memory regions that are sharing the same backing pages. When a region is aliased, `gpu_mappings` is incremented:

```
u64 kbase_mem_alias(struct kbase_context *kctx, u64 *flags, u64
stride,
            u64 nents, struct base_mem_aliasing_info *ai,
            u64 *num_pages)
{
    ...
    for (i = 0; i < nents; i++) {
        if (ai[i].handle.basep.handle > PAGE_SHIFT) <gpu_alloc;
            ...
            kbase_mem_phy_alloc_gpu_mapped(alloc);
//gpu_mappings
        }
        ...
    }
    ...
}
```

The `gpu_mappings` is checked in the `KBASE_IOCTL_MEM_COMMIT` call to ensure that the region is not mapped multiple times:

```
int kbase_mem_commit(struct kbase_context *kctx, u64 gpu_addr,
u64 new_pages)
{
    ...
    if (atomic_read(&reg->gpu_alloc->gpu_mappings) > 1)
        goto out_unlock;
    ...
}
```

So, by overwriting `gpu_mappings` of a memory region to zero, I can cause an aliased memory region to pass the above check and have its backing store resized. This then causes its backing pages to be removed without removing the alias mappings. In particular, after shrinking the backing store, the alias region can be used to access backing pages that are already freed.

The situation is now very similar to what I had in "Corrupting memory without memory corruption" and I can apply the technique from the section, "Breaking out of the context," to this bug.

To recap, I now have a `kbase_va_region` whose backing pages are already freed and I'd like to reuse these freed backing pages so I can gain read and write access to arbitrary memory. To understand how this can be done, we need to know how backing pages to a `kbase_va_region` are allocated.

When allocating pages for the backing store of a `kbase_va_region`, the `kbase mem pool alloc pages` function is used:

```
int kbase_mem_pool_alloc_pages(struct kbase_mem_pool *pool,
size_t nr_4k_pages,
        struct tagged_addr *pages, bool partial_allowed)
{
    ...
    /* Get pages from this pool */
    while (nr_from_pool--) {
        p = kbase_mem_pool_remove_locked(pool);     //next_pool)
{
        /* Allocate via next pool */
        err = kbase_mem_pool_alloc_pages(pool->next_pool,
//<----- 2.
                nr_4k_pages - i, pages + i, partial_allowed);
        ...
    } else {
        /* Get any remaining pages from kernel */
        while (i != nr_4k_pages) {
            p = kbase_mem_alloc_page(pool);     //<------- 3.
            ...
        }
        ...
    }
    ...
}
```

The input argument `kbase_mem_pool` is a memory pool managed by the `kbase_context` object associated with the driver file that is used to allocate the GPU memory. As the comments suggest, the allocation is actually done in tiers. First the pages will be allocated from the current `kbase_mem_pool` using `kbase mem pool remove locked` (1 in the above). If there is not enough capacity in the current `kbase_mem_pool` to meet the request, then

`pool->next_pool` is used to allocate the pages (2 in the above). If even `pool->next_pool` does not have the capacity, then `kbase_mem_alloc_page` is used to allocate pages directly from the kernel via the buddy allocator (the page allocator in the kernel).
When freeing a page, the same happens: `kbase_mem_pool_free_pages` first tries to return the pages to the `kbase_mem_pool` of the current `kbase_context`, if the memory pool is full, it'll try to return the remaining pages to `pool->next_pool`. If the next pool is also full, then the remaining pages are returned to the kernel by freeing them via the buddy allocator.
As noted in "Corrupting memory without memory corruption," `pool->next_pool` is a memory pool managed by the Mali driver and shared by all the `kbase_context`. It is also used for allocating page table global directories (PGD) used by GPU contexts. In particular, this means that by carefully arranging the memory pools, it is possible to cause a freed backing page in a `kbase_va_region` to be reused as a PGD of a GPU context. (The details of how to achieve this can be found in the section, "Breaking out of the context.") As the bottom level PGD stores the physical addresses of the backing pages to GPU virtual memory addresses, being able to write to a PGD allows me to map arbitrary physical pages to the GPU memory, which I can then read from and write to by issuing GPU commands. This gives me access to arbitrary physical memory. As physical addresses for kernel code and static data are not randomized and depend only on the kernel image, I can use this primitive to overwrite arbitrary kernel code and gain arbitrary kernel code execution.
The exploit for Pixel 6 can be found here with some setup notes.

Conclusions

In this post I've shown how root cause analysis of CVE-2022-36449 revealed the unusual memory management in `KBASE_MEM_TYPE_IMPORTED_USER_BUF` memory region, which then led to the discovery of another vulnerability. This shows how important it is to carry out root cause analysis of existing vulnerabilities and to use the knowledge to identify new variants of an issue. While CVE-2022-46395 seems very difficult to exploit due to a very tight race window and the limited write primitive that can be achieved by the bug, I've demonstrated how techniques from Racing against the clock—hitting a tiny kernel race window can be

used to exploit seemingly impossible race conditions, and how UAF in memory pages can be exploited reliably even with a very limited write primitive.

How to automate a Microsoft Power Platform deployment using GitHub Actions

Low-code has enabled not only developers to deploy code more easily, but it has also lowered the barrier to entry for many others to deploy applications without having to have a large amount of coding knowledge. Low-code has enabled so many to deliver applications and solve problems in their organizations with greater ease.

Microsoft Power Platform is a powerful suite of tools that allows users to build custom solutions for their organization with low-code tools. GitHub, on the other hand, is a powerful web-based platform that allows developers to manage, secure, and deliver their code. In this blog post, we will explore how to automate a Power Platform deployment using the CI/CD capability of GitHub Actions.

What is GitHub Actions

GitHub Actions is a powerful automation tool that allows developers to automate tasks, such as building, testing and deploying code. It is actually more than just a continuous integration and continuous deployment tool. It provides a wide range of pre-built actions, which are reusable units of code that can be combined to create workflows. GitHub Actions also supports custom actions, which can be built and shared across teams and the community.

Why automation

Why automation? Simply, human error. We all make mistakes. We want to increase our efficiency and productivity, all while reducing our human errors. GitHub Actions are an easy to use automation tool that work directly from a GitHub repository, enabling your deployments to occur closer to your code. Also, there are a lot of platform tools from GitHub that you can

integrate into your code base, for example, GitHub Advanced Security. It is all about bringing your tools closer to the developer and enabling you to do more with your code.

Automating your Power Platform deployment with GitHub Actions

Automating your Power Platform deployment with GitHub Actions has several benefits:

1. **Consistency**: automation ensures that each deployment is identical and prescriptive, eliminating the risk of human error.
2. **Efficiency**: reduces the time and effort required to deploy solutions, allowing you and your teams to deliver more value at an increased rate.
3. **Version Control**: track your changes with GitHub as your version control provider, allowing you to have full visibility of code changes and the ability to rollback to a previous version if needed, preventing unnecessary downtime.
4. **Collaboration**: leveraging GitHub to allow your teams to work on the same code simultaneously, increasing communication across your teams.

Watch the video below to follow along with a hands-on tutorial:

In the above video we cover off everything you need to know about deploying your Power Platform deployment with GitHub Actions; follow the links below to jump into each topic:

- Integrate Power Platform development into GitHub Codespaces
- Leverage GitHub Environments to segment your workflow
- Access the Microsoft Power Platform actions lab from a GitHub repository
- Export your Power Platform solution and check it in
- Release your Power Platform solution from a QA environment through to production
- Review your CI/CD deployment in Power Platform

Automating your Power Platform deployments with GitHub Actions can help you save time, reduce errors, and increase productivity. Good luck automating and remember, you can automate so much more than your CI/CD workflows with GitHub Actions!

Useful Resources

- Further documentation for GitHub Actions and Power Platform
- GitHub Marketplace for Power Platform Actions
- Getting started with GitHub Actions
- Learn more about Microsoft Power Platform

Kelsey Hightower on leadership in open source and the future of Kubernetes

Subscribe to The ReadME Podcast on Apple Podcasts, Spotify, or wherever you listen to podcasts.

“The future of Kubernetes, if we’re being honest, is that it has to go away. If we’re still talking about Kubernetes 20 years from now, that would be a sad moment in tech because we didn’t come up with any better ideas,” says Kelsey Hightower, Kubernetes superstar and developer advocate.

However, Kubernetes isn’t going anywhere just yet, and Kelsey outlines the unique features that keep the community moving forward. “Kubernetes has enough extension points for security, storage modules, cloud provider integrations, and so on, that there’s no need to hit the fork button. If you’ve been a maintainer, you know that the hardest thing is to add new functionality without getting sidetracked. Kubernetes’ API model, its plugin model, was a gift to all future maintainers. It relieves this group from having to figure out how to add every bell and whistle.”

In this special episode of The ReadME Podcast, dedicated to GitHub’s Maintainer Month, Kelsey joins hosts Martin Woodward and Neha Batra to discuss his philosophy on fostering thriving open source communities and the importance of empathy to a maintainer’s success. The conversation underscores the critical role of the individuals behind the open source projects we consume and celebrates their tireless efforts and the profound impact they’ve had on the tech community.

Thank you to Aaron Francis, Cassidy Williams, Frances Coronel, Anthony Sottile, Peter Strömberg, and Brandon Ringe, and of course, Kelsey Hightower, for helping us to celebrate this Maintainer Month!

Kelsey Hightower	In this special episode, Kelsey shares his origin story, insights on the future of Kubernetes, and advice on making

—Present	complicated technology easier to understand.
The open/closed equilibrium	Striking a balance between openness and control in open source projects, preserving the integrity of community insights, and how humor can transform communities.
Fusing tech and progress	How open source is powering nuclear fusion research, advice for fortifying your career against change, and practical tips on using GitHub.
Innovation without barriers	Reframing disability and accessibility, playing Minecraft with your eyes, and what AI means for the future of accessibility.

To hear all of Kelsey's advice, including tips on succession planning and how to identify future project leaders, tune in to this bonus episode of The ReadME Podcast. And don't miss next month's episode, where we'll go beyond the code to examine what it takes to build a successful open source project. Subscribe to The ReadME Podcast on Apple Podcasts, Spotify, or wherever you listen.

Announcing the public preview of GitHub Advanced Security for Azure DevOps

Web applications are foundational to nearly every aspect of everyday life, whether they are used for shopping and remote work, or to provide life-saving services in hospitals and power critical infrastructure. However, the proliferation of web applications doesn't come without risk. Applications continue to be a top attack vector, and are at the center of more than 40% of all data breaches.

At GitHub, we want to make it as easy as possible to not only build innovative software, but build it securely. GitHub Advanced Security's (GHAS) application security testing tools were built to provide a frictionless, native experience for developers, to help drive innovation forward. This native approach is critical, as oftentimes security findings take six months or more to fix. With GHAS' real time vulnerability detection, developers can fix issues in minutes, not months. For instance, the fix rate of vulnerabilities identified by CodeQL during a pull request is 72% compared to the industry norm fix rate of 15%, seven days after a vulnerability has been detected. This is just one of the reasons GHAS users fixed 24 million vulnerable packages in 2022.

Today, GHAS will be publicly available on Azure DevOps. GHAS has been a game-changer for many development teams, providing critical application security testing capabilities, such as secret scanning, dependency scanning (SCA), and code scanning (SAST) natively in the developer workflow. With these features natively embedded in Azure DevOps, teams can leverage the power of GHAS without leaving their familiar Azure DevOps environment.

Secret scanning: stop secret leaks

Secret scanning detects and prevents secret exposure in your application development process. Stolen credentials are present in nearly 50% of security incidents, highlighting the need for organizations to secure their secrets. GHAS for Azure DevOps provides out-of-the-box secret scanning, with no additional tooling required. You can easily enable it on all your repositories to instantly detect exposed secrets. In 2022 alone, GitHub detected over 1.7 million exposed secrets.

Dependency scanning: secure your software supply chain

Dependency scanning is another key feature that can help identify vulnerabilities in open source packages used in Azure Repos. With the rise of open source supply chain attacks, and the presence of vulnerabilities like Log4Shell, developers need to take extra precautions to ensure their code is secure. GHAS for Azure DevOps identifies the open source packages used in Azure Repos and provides guidance on how to upgrade those packages to mitigate vulnerabilities.

Code scanning: prevent and fix vulnerabilities in your code

Code scanning is a critical component of any robust application security strategy, and GHAS' CodeQL static analysis engine has quickly become an industry leader in detecting static code vulnerabilities. With the integration of CodeQL scans directly into Azure Pipelines, developers can now detect hundreds of code security vulnerabilities across a wide range of languages, including C#, C/C++, Python, JavaScript/TypeScript, Java, Go, and more.

Interested in learning more? Sign up for the preview, and we'll do our best to get your Azure DevOps organization(s) enabled as soon as possible!.

---

devops/