挖洞思维--思想建设

最近也没有挖到什么漏洞，分析类的现在卡的也严格，想了想，总结一些关于漏洞的看法吧。
为什么我要谈这些呢？很多年轻小伙子肯定觉得：“唉，又是老年人回顾过去的辉煌，不就挖了几个漏洞么就开始写经验总结了？大道理一套一套的，没个屁用。”确实，对于一个急于求rce的人来说，这些东西太虚太飘渺不如一个exp.py来的直接有用。
但如果读者真心追求真理的窥探，想要成为一台真正的漏洞挖掘机，那么方法论其实是很重要的，这是一种思维模式，而思维模式在很多情况下要比具体的技术原理对你帮助要大。

0x00 分类分层

我的认知里面一般将漏洞（或者说安全问题）分为几个层次：

1. 协议原理层面，即规范定义、设计，设计之初就存在缺陷
2. 代码实现层面，即对设计的实现，不同产品的实现会存在差异
3. 使用配置层面，即对单一的组件使用配置的时候，配置的问题导致存在问题
4. 多组件配合层面，即多个组件之间配合使用时候存在问题导致问题
5. 基础设施交互层面，即和操作系统、网络之间交互时导致的问题
6. 实际运维层面，即代入到实际环境中，引入人的行为作为整体的一部分带来的隐患

其实也比较难把所有东西都归纳进去，但基本上可以概括绝大部分场景了。这么分层的意义主要在于避免自己的视野被局限在某个层面，因为我们很容易在自己擅长的层面疯狂的卷当卷不出来的时候就会迷茫，一旦视野被局限了，就可能忽视了一些问题，要知道一个非常重要的事实：
任何系统都不是跑在一个真空环境中
我们以某java cms审计为例，通常在审计这种cms的时候，我们会在代码里寻找危险函数，审计各种上传漏洞、反序列化、XXE、命令注入、越权等。很显然这种审计挖掘工作在“代码实现层面”，那么当这个层面的漏洞被卷完了，一滴不剩了，我们该做什么?
这种事情很常见，那这时候我们是不是要跳出cms自身的层面，来看看他所使用的框架、组件，比如Spring框架自身的一些配置问题，框架的版本缺陷，如果他使用了docker，攻击面还能扩大，比如官方通用的docker启动时候是否携带了默认配置导致一系列问题。
跳出固有层面的思维是很重要的，哪怕是同一个“代码实现层面”，从一个java函数跳到该函数底层C实现，这也是层面思维的突破。

0x01 避免原子性思想

什么叫原子性思想?
这个很常见，最近群里讨论了java的getruntime.exec执行命令的问题就是典型的原子性思想。很多人在遇到这个函数的时候，都会直接把它当作一个执行shell命令的函数。而很少有人真正理解这个函数如何实现，那么这会导致什么问题呢?
当我们把某个底层函数看成是原子的，认为它已经是底层了，不可分割了，就像前面说的exec一样，在大部分人看来它已经是java层面上的一个原子，无法再突破了。
那么当我们深入理解其实现的时候发现，它其实根本不简单，从java层一直到C层，C层最后是系统调用。你看，当我们不再把它看成一个原子的时候就能突破到C层面，那么问题来了，系统调用是原子了吗？并不是，系统调用依旧是代码写的，他还能突破，具体可以去看linux源码，这里面还能发现更多的细节，而这些细节可能会引入新的攻击面，毕竟查了查，Linux内核低版本的时候确实出现过exec*的CVE。
不仅仅是函数调用上我们容易犯原子化思想的错误，我们再来看看一个场景：
`system("ping 'xxxxxxxxx'")`
这样一个命令执行，假设xxx可控，并且过滤了一切你可能逃逸或者嵌入命令的东西，那是不是没救了？不一定。
这里我们又犯了一个错误，那就是把ping命令原子化了，认为ping命令实现非常完美，完美到变成一个原子。这时候我们尝试把ping这个原子分割，进一步潜入其实现，那么就会出现一些其他的可能攻击面：

1. ping本身在处理畸形数据的时候可能存在漏洞，比如溢出CVE等
2. 控制ping来处理我们可控服务的返回畸形ICMP返回包时其处理有问题导致漏洞

如果面对的是一个网络设备，那么通常系统版本都是固定的，那么这个ping的版本是可以确认的，这时候去找到这个ping的实现代码开始审计，同时查一查公开的CVE，这时候可能就会有nday或者0day出现了。
当然这不是说ping一定有漏洞，这里主要讲的还是一种思维。

0x02 功能第一安全第二

我们很多安全从业人员做安全做久了就容易眼里只盯着漏洞，看不到其他业务的功能。实际上在任何一个组件（甚至是函数）他都有自己的实现功能目的，这些功能大部分时候是服务人类实际的需求。这里说的业务功能并不是表层的业务增删改查，而是指所有运行中的实现（比如系统内存分配、缓存使用、性能优化算法等等）
为什么我要说这个呢？这听起来和挖洞没关系。其实不然，当我们意识到任何组件实际出发点肯定是先实现自己的功能，那么它出现漏洞是必然。这里我以fastjson为例，我们都嘲笑fastjson漏洞很多，却没有人真正明白为什么它有很多漏洞，是技术问题吗？并不是，本质上原因还是

得从功能出发，我们思考一下为什么它要叫做fastjson?
从名字中我们就可以看到开发它的初心，那就是为了实现“fast”，这个fast很多人以为是性能fast，其实不是，是开发效率fast！程序员总是懒惰的，如果一个组件能让他们少写两行代码他们肯定吹飞起来。fastjson用起来方便顺滑，所以慢慢的开始流行起来。
我们现在明白了它的立意后，就能明白为什么它会有很多漏洞，原因很简单，就是因为它太无脑了，无脑的背后是大量的灵活特性，各种动态映射。
为什么它一直修一直有漏洞？是因为开发者技术不行吗？其实也不是，还是从功能角度出发，如果fastjson干掉了那些特性，那他可以直接改名slowjson了，不可能因为几个安全漏洞而把自己初衷直接干翻，所以只能在保留fast的基础上修修补补。我敢直接说，如果为了杜绝漏洞，开发者把所有灵活特性干掉，那么他最后会直接放弃维护，因为这违背了他的价值观也违背了产品定位。
回过头来看漏洞，一个组件如果稳定在某个版本，并且这个版本的漏洞常年挖掘不到了，那么该版本的这个组件一定很安全。相反的，如果它一天到晚升级，那么肯定是不安全，为什么？因为大部分升级都是为了添加新功能新特性让自己变得更好用，升级修复高危漏洞其实是顺带的。只要有新功能新特性则必然带入新漏洞。

0x03 系统是动态的

我们在挖掘漏洞的时候一定要把自己代入到实际环境中，明白系统是动态运行的。
我知道很多人在审计代码的时候，往往只是跟着source一步步到sink，卡住了就是卡住了，这条路径不通换一条路径，这感觉就像走迷宫一样。这样做确实没有问题，也是正常的，但当我们发现所有路径都不通的时候，就要意识到我们陷入了实验室逻辑里。这时候跳出思维是非常必要的，把思维跳出来，自己代入到场景里，意识到系统本身在动态运行。他受到内存、CPU、系统自身上的任何东西运行限制，把你的目标放入到一个真实的带有时间轴的系统中，它的周围甚至还有很多乱七八糟的应用和运维脚本，这时候，会不会出现新的攻击面？不好说，但是值得尝试。

0x04 大部分实现只是约等于

很多功能实现本质上是不完美的，这种不完美甚至是超出你的想象，但大部分不一定会直接导致一个明显的漏洞。我们要经常性的去搜集一些细微的差异，这种差异有时候比直接的漏洞更具备技术魅力，因为他们会创造一些可能性，这些可能性在遇到合适的场景进行串联的时候，就能把不可能变成可能。
为什么存在这么多不完美的实现呢？原因很简单，实际场景下开发者的思维总是有限的，他看不到所有的可能性，因此也只能写出有限条件的判断，那么超出这些常见条件的判断就难以处理。还有一种非常常见的原因就是兼容性。为什么要兼容？因为复杂的系统通常是N个人开发的，开发能力不对齐，前人埋坑后人填，后人为了能让功能正常跑不得不加入一些诡异的逻辑让自己的程序去兼容前者。同一个程序里也有不同时期的人写的补丁。所以日积月累就会有一大堆和预期不太符合但是又能奇妙跑起来的feature。
找到这些特性，比找到漏洞带来的价值会更高，因为它们隐藏的深而且带来的可能性多。

0x05 凡人要专注

年轻人总是心高气傲，一脸我就是天下第一绝顶聪明大天才的样子。
我觉得不管你是谁，哪怕是真的天才，也要把自己当作蠢材，我认为没有人是真正的天才。只要智力没有缺陷大家大体上是差不多的，只不过就是专注度问题。
如果你挖不到漏洞，首先想想自己是不是足够专注。专注在某一个领域，专注在某一个具体产品，不断投入时间，你肯定能挖到漏洞，最多就是别人用一天时间挖一个rce，而你需要用一个月。但是，你依旧可以挖到。
凡人，这样就够了。

0x06 光学习不挖洞，光挖洞不学习，都是废物

有些人就跟学习机器一样，学了非常多的文章，却从没有产出，更像一个学者而不是实际研究员。我觉得这样非常不好，你复现再多漏洞，你看过再多文章，每个链倒背如流，那也是背的别人的东西，你自己去操作，去挖掘，完全是两码事。如果学习只是学习知识，一辈子都是在往脑子里装知识却没有把知识转化成实际的结果。那么站在唯物主义的角度来看，你学了跟没学一样，这辈子是对外界来说是没有区别的，甚至可以说你浪费一辈子的时间在学习上还不如去打游戏。所以一定要强调实际产出。
同理，很多人天天挖洞，不学习新的思路，挖来挖去那点操作，挖完了就换一个设备继续挖，那我说实话，这就是台挖掘同类型漏洞的机器，长期不突破只是横向跨越到不同的产品上，这样早晚要被AI替代掉。所以阶段性学习新东西使自己突破到不新的高度是非常重要的。

0x07 不是一定要学完所有东西才能挖到洞

很多人学习起来挺厉害的，但是什么都想学，什么热学什么，这块不足补一下，那块不足补一下，每天那么多新技术新思路出来，你天天跟着学，学得完吗?
对大部分常见漏洞类型的原理有理解，知道这些漏洞会出现在哪些点，它们能做到什么效果，这些知道其实就够了，具体如何实施某漏洞的攻击技巧，完全可以在遇到这个漏洞点后需要深入利用时再去边学边上手。技巧是学不完的，极端技巧依赖于环境，你这辈子遇不到可以完全不用学。

0x99 先聊这些

先说这些吧，后面再聊聊更加具体一点的方法论。这篇主要是思想建设，后面想到了别的继续补上。