游戏服务器开发避坑&&内存安全编程实践

游戏服务器开发是一项复杂的工作，涉及网络编程、并发处理、数据存储等多个方面，同时也面临着各种挑战和潜在的“坑”。本次分享旨在介绍一些经验技巧，帮助大家避开开发中一些常见的问题，提高开发质量，写出更可靠的代码。

软件缺陷统计

2023年最危险软件缺陷前25名
CWE Top 25 Most Dangerous Software Weaknesses

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2022
1	CWE-787	Out-of-bounds Write	63.72	70	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.54	4	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	34.27	6	0
4	CWE-416	Use After Free	16.71	44	+3
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	15.65	23	+1
6	CWE-20	Improper Input Validation	15.50	35	-2
7	CWE-125	Out-of-bounds Read	14.60	2	-2
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.11	16	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.73	0	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.41	5	0
11	CWE-862	Missing Authorization	6.90	0	+5
12	CWE-476	NULL Pointer Dereference	6.59	0	-1
13	CWE-287	Improper Authentication	6.39	10	+1
14	CWE-190	Integer Overflow or Wraparound	5.89	4	-1
15	CWE-502	Deserialization of Untrusted Data	5.56	14	-3
16	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	4.95	4	+1
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.75	7	+2
18	CWE-798	Use of Hard-coded Credentials	4.57	2	-3
19	CWE-918	Server-Side Request Forgery (SSRF)	4.56	16	+2
20	CWE-306	Missing Authentication for Critical Function	3.78	8	-2
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.53	8	+1
22	CWE-269	Improper Privilege Management	3.31	5	+7
23	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.30	6	+2
24	CWE-863	Incorrect Authorization	3.16	0	+4
25	CWE-276	Incorrect Default Permissions	3.16	0	-5

已知被利用的漏洞前10名

CWE Top 10 KEV(Known Exploited Vulnerabilities) Weaknesses

KEV Weaknesses Rank	CWE-ID	Weakness Name	Analysis Score	Number of Mappings in the KEV Dataset	Average CVSS
1	CWE-416	Use After Free	73.99	44	8.54
2	CWE-122	Heap-based Buffer Overflow	56.56	32	8.79
3	CWE-787	Out-of-bounds Write	51.96	34	8.19
4	CWE-20	Improper Input Validation	51.38	33	8.27
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	49.44	25	9.36
6	CWE-502	Deserialization of Untrusted Data	29.00	16	9.06
7	CWE-918	Server-Side Request Forgery (SSRF)	27.33	16	8.72
8	CWE-843	Access of Resource Using Incompatible Type ('Type Confusion')	26.24	16	8.61
9	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	19.90	14	8.09
10	CWE-306	Missing Authentication for Critical Function	12.98	8	8.86

缺陷案例

不合法的输入

案例1

```
double price = 20.00;
int count = buyReq.buy_count;  //buyReq.buy_count = -1
double total = price * count;
user.sub_money(total);
user.add_goods_cnt(count);
```

客户端请求时输入的count是负数，在缺少验证的情况下，可能导致玩家购买物品扣负数的钱。

案例2

```
int count = buyReq.get_reward_count;
user.add_goods_cnt(count);
```

设计错误，领取奖励的数量直接让客户端发上来，且没做验证。

案例3 SQL注入

```
string userName = ctx.getAuthenticatedUserName();
string query = "SELECT * FROM items WHERE owner = '" + userName + "' AND itemname = '" + ItemNar
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
```

ItemName由客户端输入，如果缺少验证，其输入内容为name' OR '1'='1 甚至是;delete from items，就会导致SQL注入，发生严重安全问题。

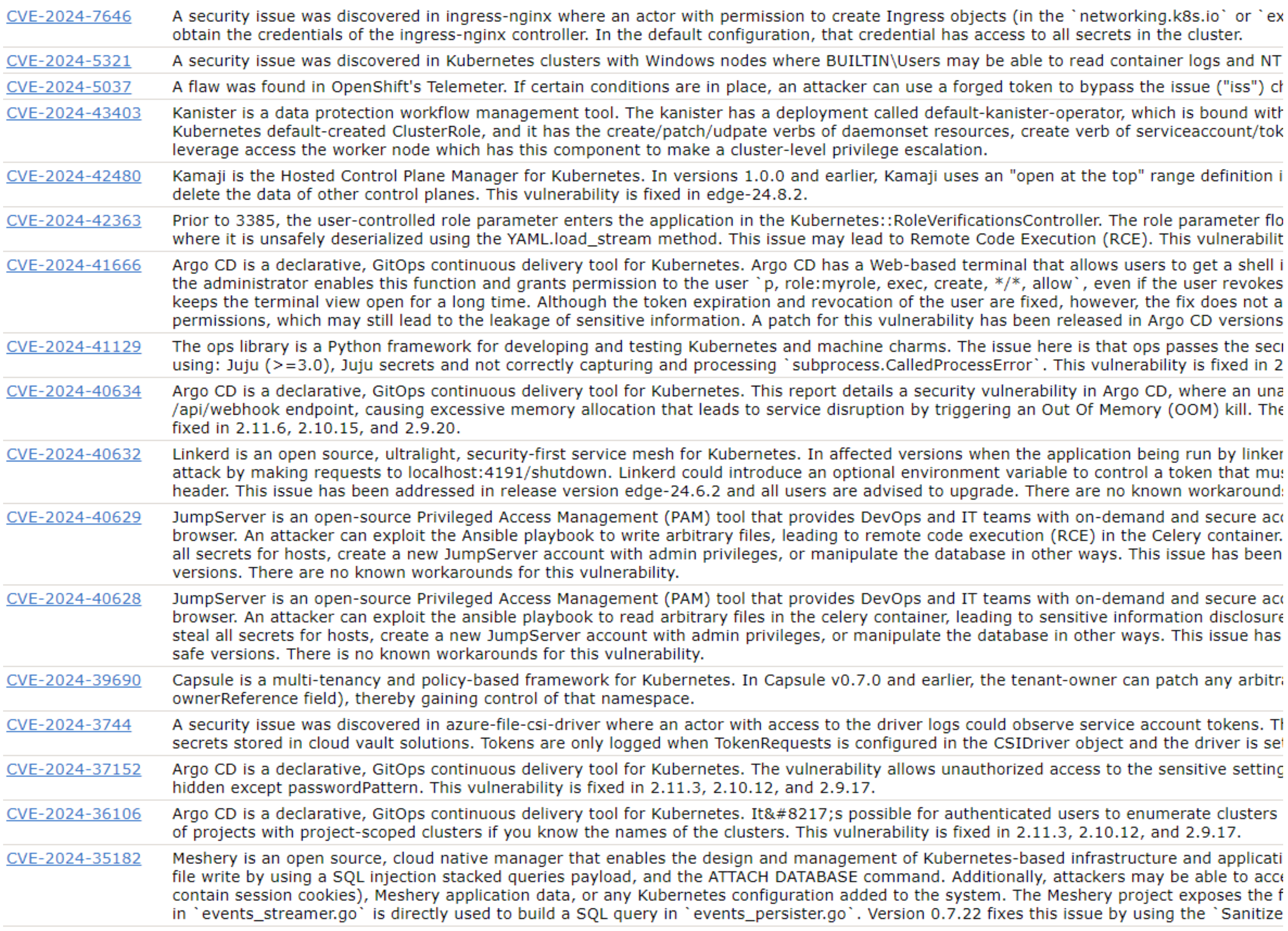
其他案例

聊天系统发广告，发不文明词汇。
频繁发送消耗服务器性能的请求，比如批量拉取其他玩家公开信息等。
发送伪造的数据包攻击服务器。

解决方案：对所有外部输入，默认都采用不信任的方式，**必须**严格验证其合法性，确保输入在设计要求的范围内。

权限问题

kubernetes历史漏洞



CVE-2024-7646	A security issue was discovered in ingress-nginx where an actor with permission to create Ingress objects (in the `networking.k8s.io` or `ex obtain the credentials of the ingress-nginx controller. In the default configuration, that credential has access to all secrets in the cluster.
CVE-2024-5321	A security issue was discovered in Kubernetes clusters with Windows nodes where BUILTIN\Users may be able to read container logs and NT
CVE-2024-5037	A flaw was found in OpenShift's Telemeter. If certain conditions are in place, an attacker can use a forged token to bypass the issue ("iss") cl
CVE-2024-43403	Kanister is a data protection workflow management tool. The kanister has a deployment called default-kanister-operator, which is bound witl Kubernetes default-created ClusterRole, and it has the create/patch/udpate verbs of daemonset resources, create verb of serviceaccount/tok leverage access the worker node which has this component to make a cluster-level privilege escalation.
CVE-2024-42480	Kamaji is the Hosted Control Plane Manager for Kubernetes. In versions 1.0.0 and earlier, Kamaji uses an "open at the top" range definition i delete the data of other control planes. This vulnerability is fixed in edge-24.8.2.
CVE-2024-42363	Prior to 3385, the user-controlled role parameter enters the application in the Kubernetes::RoleVerificationsController. The role parameter flo where it is unsafely deserialized using the YAML.load_stream method. This issue may lead to Remote Code Execution (RCE). This vulnerabilit
CVE-2024-41666	Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. Argo CD has a Web-based terminal that allows users to get a shell i the administrator enables this function and grants permission to the user `p, role:myrole, exec, create, \*/\*, allow`, even if the user revokes keeps the terminal view open for a long time. Although the token expiration and revocation of the user are fixed, however, the fix does not a permissions, which may still lead to the leakage of sensitive information. A patch for this vulnerability has been released in Argo CD versions
CVE-2024-41129	The ops library is a Python framework for developing and testing Kubernetes and machine charms. The issue here is that ops passes the secı using: Juju (>=3.0), Juju secrets and not correctly capturing and processing `subprocess.CalledProcessError`. This vulnerability is fixed in 2
CVE-2024-40634	Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. This report details a security vulnerability in Argo CD, where an una /api/webhook endpoint, causing excessive memory allocation that leads to service disruption by triggering an Out Of Memory (OOM) kill. The fixed in 2.11.6, 2.10.15, and 2.9.20.
CVE-2024-40632	Linkerd is an open source, ultralight, security-first service mesh for Kubernetes. In affected versions when the application being run by linker attack by making requests to localhost:4191/shutdown. Linkerd could introduce an optional environment variable to control a token that mu: header. This issue has been addressed in release version edge-24.6.2 and all users are advised to upgrade. There are no known workaround:
CVE-2024-40629	JumpServer is an open-source Privileged Access Management (PAM) tool that provides DevOps and IT teams with on-demand and secure acı browser. An attacker can exploit the Ansible playbook to write arbitrary files, leading to remote code execution (RCE) in the Celery container. all secrets for hosts, create a new JumpServer account with admin privileges, or manipulate the database in other ways. This issue has been versions. There are no known workarounds for this vulnerability.
CVE-2024-40628	JumpServer is an open-source Privileged Access Management (PAM) tool that provides DevOps and IT teams with on-demand and secure acı browser. An attacker can exploit the ansible playbook to read arbitrary files in the celery container, leading to sensitive information disclosure steal all secrets for hosts, create a new JumpServer account with admin privileges, or manipulate the database in other ways. This issue has safe versions. There is no known workarounds for this vulnerability.
CVE-2024-39690	Capsule is a multi-tenancy and policy-based framework for Kubernetes. In Capsule v0.7.0 and earlier, the tenant-owner can patch any arbitrı ownerReference field), thereby gaining control of that namespace.
CVE-2024-3744	A security issue was discovered in azure-file-csi-driver where an actor with access to the driver logs could observe service account tokens. Tl secrets stored in cloud vault solutions. Tokens are only logged when TokenRequests is configured in the CSIDriver object and the driver is se
CVE-2024-37152	Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. The vulnerability allows unauthorized access to the sensitive setting hidden except passwordPattern. This vulnerability is fixed in 2.11.3, 2.10.12, and 2.9.17.
CVE-2024-36106	Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. It&#8217;s possible for authenticated users to enumerate clusters of projects with project-scoped clusters if you know the names of the clusters. This vulnerability is fixed in 2.11.3, 2.10.12, and 2.9.17.
CVE-2024-35182	Meshery is an open source, cloud native manager that enables the design and management of Kubernetes-based infrastructure and applicati file write by using a SQL injection stacked queries payload, and the ATTACH DATABASE command. Additionally, attackers may be able to acce contain session cookies), Meshery application data, or any Kubernetes configuration added to the system. The Meshery project exposes the f in `events_streamer.go` is directly used to build a SQL query in `events_persister.go`. Version 0.7.22 fixes this issue by using the `Sanitize

案例1

越权访问其他用户数据。
如游戏设计了一个查询其他玩家公开信息的接口，但是里面包含了私有的敏感信息（如账号）。或者接口参数不是公开的，返回的信息也是私有的，但是参数可以猜测。

```
message GetUserDetailInfoReq{
  uint64 uid = 1;
}

message GetUserDetailInfoRep{
  string account = 1;
  ...
}
```

案例2

在没有权限的情况下，违规修改其他用户数据。
如因没有做好权限设计和检查，让普通用户调用了GM功能。
如代码实现设计错误，一些应该是只读的功能，如查询玩家的信息，调用了会修改、保存玩家数据的接口。

解决方案：

- 在查询数据之前，验证当前用户是否有权限访问请求的数据。
- 实现基于角色的访问控制或其他权限管理系统来管理用户权限。
- 最小权限原则：确保用户只能访问他们自己的数据。对A用户修改B用户数据的功能要提高警惕。

类型安全问题

案例1

```
int price = 2;
int count = max_int;
int total_money = price * count;
user.sub_money(total_money);
user.get_goods(count);
```

price * count溢出了，减了错误的钱。

案例2

```
//判断要减的货币数量是否合法
unsigned int get_total_money(int price, int count)
{
  return price * count;
}

int price = 2;
int count = max_int;
auto total_money = get_total_money(price,count);

if(total_money > 0)
{
  if(user.sub_money(total))
  {
    user.get_goods(count);
  }
}
```

因为总价被误转换为unsigned int，导致后面的total_money>0的校验逻辑失效。

硬编码

案例1

```
bool IsAdmin(char *password) {
if (strcmp(password, "whosyourdaddy")==0) {
   return true;
}
...
return false;
}
```

硬编码对代码后续的维护非常不友好，很容易产生各种问题。

并发问题 Data Race & Race Condition

并发编程会产生一系列复杂性。避免内存读写时的Data Race，我们需要做到下面四点：

- 共享数据，但只读共享

- 不共享数据（Actor模型）
- 如果共享数据被读和写，那么要保护好共享的数据（Atomic，Mutex，RWLock。。。）
- 保证数据被访问的时候是有效的

而Race Condition情况往往更加复杂，必须综合整段逻辑进行考虑。

案例1

```go
func buyItem(p *Player) {
  if p.YuanBao >= item.Cost {
    p.YuanBao -= item.Cost
    p.ItemCount++
    }
}

func main() {
  player := Player{}
  go buyItem(&player)
  go buyItem(&player)
}
```

改为

```go
var mu = sync.Mutex{}
func buyItem(p *Player) {
  mu.Lock()
  defer mu.Unlock()
  if p.YuanBao >= item.Cost {
    p.YuanBao -= item.Cost
    p.ItemCount++
    }
  }
```

内存管理

内存管理主要有以下方式:

- **C/C++ - 手动管理 + 部分自动管理**

    - 手动管理: C和C++提供了直接的内存管理能力，通过malloc, calloc, realloc, 和free（C语言）或new和delete（C++）来分配和释放内存。这种方式灵活但容易出错，需要程序员自行管理内存生命周期，以避免内存泄漏和悬挂指针等问题。
    - 智能指针：C++引入了智能指针（如std::unique_ptr, std::shared_ptr）作为一种自动管理内存的方式，通过RAII（Resource Acquisition Is Initialization）机制自动管理内存生命周期，减少手动管理的负担。
- **Java - 自动垃圾回收（Garbage Collection, GC）**
  Java使用自动垃圾回收机制来管理内存，程序员不需要直接参与内存的分配和释放。Java虚拟机（JVM）的垃圾收集器负责追踪对象的引用，自动回收不再使用的对象所占的内存。Java有多种GC算法，如分代收集、增量收集等，以平衡性能和资源利用率。
- **Python - 自动垃圾回收 + 引用计数**
  Python同样采用自动垃圾回收机制，主要依靠引用计数和周期检测器（如标记-清除或标记-压缩算法）。每个对象有一个引用计数，当计数变为零时，对象被视为垃圾并被回收。对于循环引用，Python使用周期检测器来定期发现并回收这些对象。
- **Go - 自动垃圾回收**
  Go语言使用了自动垃圾回收机制，设计上力求简单高效，减少了内存管理的复杂性。它的垃圾收集器是并发三色标记-清除算法的变体，旨在最小化垃圾回收带来的停顿时间(STW)，支持高并发服务的需求。
- **Rust - 所有权系统**
  Rust通过所有权系统和生命周期的概念来管理内存，这是一种独特的自动内存管理方式。Rust编译器在编译时强制执行所有权规则，确保内存安全，避免了内存泄漏和悬挂指针等问题。它使用借用和智能指针（如Box, Rc, Arc)来管理内存生命周期，而不需要运行时垃圾回收。

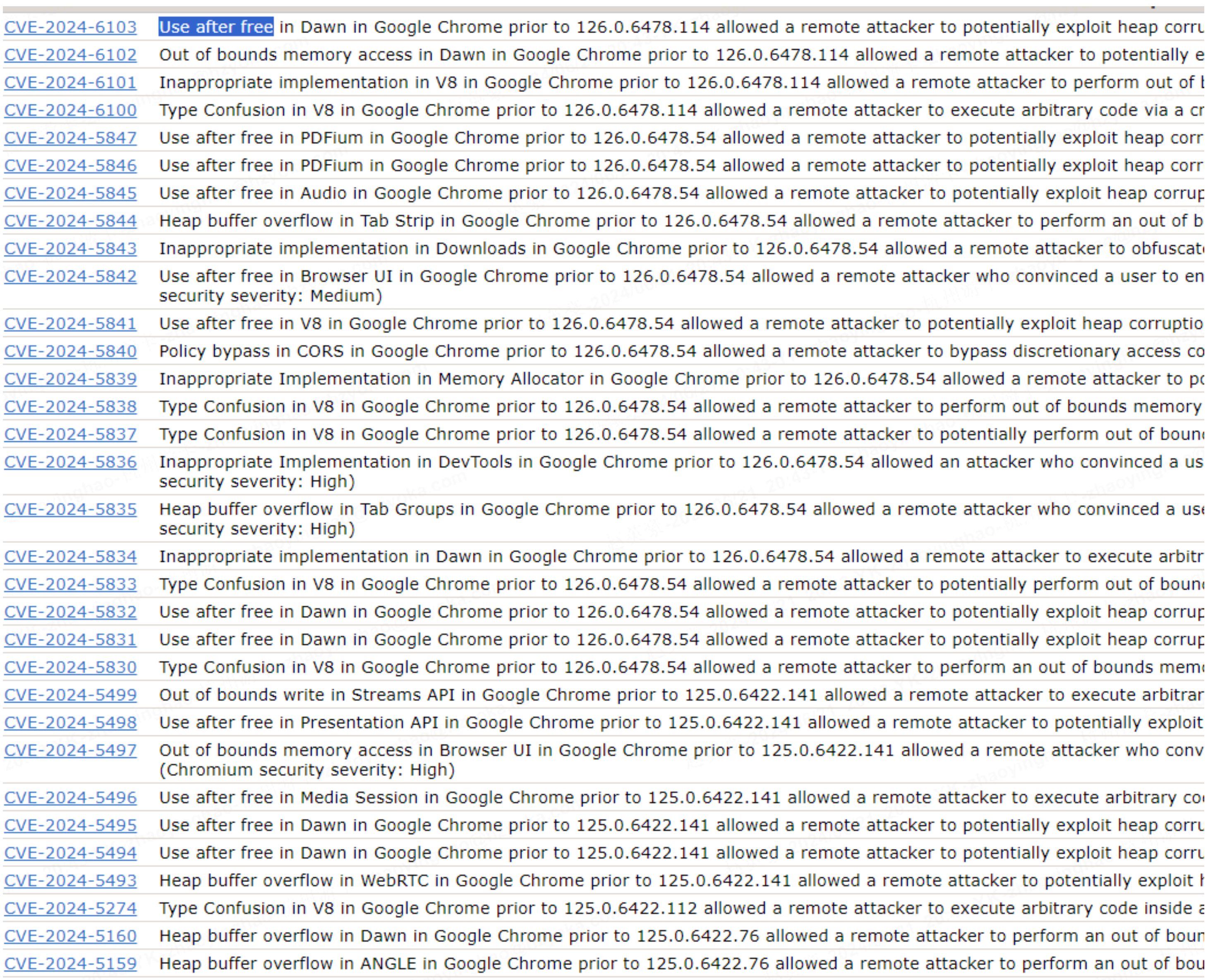
每种语言的内存管理方式都反映了其设计哲学和目标应用场景：
手动管理提供了最大的控制力和性能潜力，但需要开发者自行保证安全性，**最容易**产生各种问题；自动垃圾回收简化了内存管理，提高了开发效率，但可能在特定场景下引入性能开销；而像Rust的所有权系统则试图在两者间找到一个平衡点，提供内存安全的同时保持高性能，但需要对代码的设计实现做出约束。

对于网络服务来说，服务器大多都是基于客户端-服务器架构（C/S架构)，将用户数据集中管理，并基于网络通讯进行数据传输。而与网站等场景相比，游戏服务器有以下一些特点：

- **定制化的游戏逻辑**:游戏服务器要执行特定的游戏规则和逻辑，比如游戏事件触发、角色交互规则、得分计算等，这要求服务器软件具有高度的定制化和灵活性。
- **实时性**:为了保证游戏体验的连贯性，游戏服务器需要处理玩家的即时操作，并即时给出反馈。
- **高性能**:游戏服务器往往需要应对大量的玩家同时在线，并进行复杂的游戏逻辑运算。
- **状态同步与数据一致性**:在多人游戏中，服务器需要维护游戏世界的全局状态，并确保所有玩家客户端的状态同步，需要更强的数据一致性。

这些特点决定了游戏服务器需要尽量减少磁盘io、网络io等开销，更高效地利用cpu，把各种数据集中在内存中进行运算。因此如何使用和管理内存往往是游戏开发者绕不开的一个话题。

chrome历史漏洞



CVE-2024-6103	Use after free in Dawn in Google Chrome prior to 126.0.6478.114 allowed a remote attacker to potentially exploit heap corru
CVE-2024-6102	Out of bounds memory access in Dawn in Google Chrome prior to 126.0.6478.114 allowed a remote attacker to potentially e
CVE-2024-6101	Inappropriate implementation in V8 in Google Chrome prior to 126.0.6478.114 allowed a remote attacker to perform out of I
CVE-2024-6100	Type Confusion in V8 in Google Chrome prior to 126.0.6478.114 allowed a remote attacker to execute arbitrary code via a cr
CVE-2024-5847	Use after free in PDFium in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially exploit heap corr
CVE-2024-5846	Use after free in PDFium in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially exploit heap corr
CVE-2024-5845	Use after free in Audio in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially exploit heap corrup
CVE-2024-5844	Heap buffer overflow in Tab Strip in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to perform an out of b
CVE-2024-5843	Inappropriate implementation in Downloads in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to obfuscat
CVE-2024-5842	Use after free in Browser UI in Google Chrome prior to 126.0.6478.54 allowed a remote attacker who convinced a user to en security severity: Medium)
CVE-2024-5841	Use after free in V8 in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially exploit heap corruptio
CVE-2024-5840	Policy bypass in CORS in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to bypass discretionary access co
CVE-2024-5839	Inappropriate Implementation in Memory Allocator in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to po
CVE-2024-5838	Type Confusion in V8 in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to perform out of bounds memory
CVE-2024-5837	Type Confusion in V8 in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially perform out of boun
CVE-2024-5836	Inappropriate Implementation in DevTools in Google Chrome prior to 126.0.6478.54 allowed an attacker who convinced a us security severity: High)
CVE-2024-5835	Heap buffer overflow in Tab Groups in Google Chrome prior to 126.0.6478.54 allowed a remote attacker who convinced a use security severity: High)
CVE-2024-5834	Inappropriate implementation in Dawn in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to execute arbitr
CVE-2024-5833	Type Confusion in V8 in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially perform out of boun
CVE-2024-5832	Use after free in Dawn in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially exploit heap corrup
CVE-2024-5831	Use after free in Dawn in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to potentially exploit heap corrup
CVE-2024-5830	Type Confusion in V8 in Google Chrome prior to 126.0.6478.54 allowed a remote attacker to perform an out of bounds mem
CVE-2024-5499	Out of bounds write in Streams API in Google Chrome prior to 125.0.6422.141 allowed a remote attacker to execute arbitrar
CVE-2024-5498	Use after free in Presentation API in Google Chrome prior to 125.0.6422.141 allowed a remote attacker to potentially exploit
CVE-2024-5497	Out of bounds memory access in Browser UI in Google Chrome prior to 125.0.6422.141 allowed a remote attacker who conv (Chromium security severity: High)
CVE-2024-5496	Use after free in Media Session in Google Chrome prior to 125.0.6422.141 allowed a remote attacker to execute arbitrary co
CVE-2024-5495	Use after free in Dawn in Google Chrome prior to 125.0.6422.141 allowed a remote attacker to potentially exploit heap corru
CVE-2024-5494	Use after free in Dawn in Google Chrome prior to 125.0.6422.141 allowed a remote attacker to potentially exploit heap corru
CVE-2024-5493	Heap buffer overflow in WebRTC in Google Chrome prior to 125.0.6422.141 allowed a remote attacker to potentially exploit I
CVE-2024-5274	Type Confusion in V8 in Google Chrome prior to 125.0.6422.112 allowed a remote attacker to execute arbitrary code inside a
CVE-2024-5160	Heap buffer overflow in Dawn in Google Chrome prior to 125.0.6422.76 allowed a remote attacker to perform an out of boun
CVE-2024-5159	Heap buffer overflow in ANGLE in Google Chrome prior to 125.0.6422.76 allowed a remote attacker to perform an out of bou

常见内存问题

内存初始化问题

使用未经初始化的指针或变量，其值是不确定的，可能导致程序逻辑错误或安全问题。

案例1

```
int* ptr; // 野指针声明，未初始化
*ptr = 10; // 错误：ptr的值未知，可能指向任何位置
```

改为：

```
int* ptr=nullptr;
//...
if(ptr){
  *ptr = 10;
}
```

野指针问题，引发程序崩溃或数据损坏。

案例2

```
class A
{
  int v;
  int* p;
}
A a;
...
//直接使用a.v，a.p
```

改为

```
class A
{
  int v;
  int* p;
  A():v(0),p(nullptr){}
}
```

使用初始化列表，避免遗漏初始化操作。

申请释放不匹配

malloc free和new delete不匹配，一块内存被释放多次等，都是未定义行为，可能导致程序崩溃或数据损坏。

案例1

```
struct A
{
int v;
vector<int> vec;
}

A* a = malloc(sizeof(A));
//a->vec.size(), error
```

改为

```
A* a = new A();
```

案例2

```
class A
{
public:
  A():{
    p = new int(0);
  };
  ~A(){
    delete p;
  };
  void clear(){
    delete p;
  }
private:
  int* p;
}
A a;
a.clear();
```

改为

```
~A(){
  if(p){
    delete p;
  }
};
void clear(){
  delete p;
  p = nullptr;
}
```

解决方案

- 遵循RAII原则。
- 释放内存后，立即将指针置为nullptr，避免重复释放。

内存越界写入

写入数组或动态分配的内存块超出了其边界，可能覆盖相邻内存区域的数据，导致数据损坏或安全漏洞，如栈溢出、堆溢出等。这是一种**非常严重**的漏洞。

案例1

```
int main() {
    int array[5] = {1, 2, 3, 4, 5};
    array[5] = 10;
    return 0;
}
```

案例2

```
void printMessage(char* message) {
    char buffer[16];
    strcpy(buffer, message);
    printf("Your message: %s\n", buffer);
}

int main() {
    char largeMessage[30] = "looooooooooooooooong msg";
    printMessage(largeMessage);
    return 0;
}
```

改为

```
strncpy(buffer, message, sizeof(buffer) - 1);
buffer[sizeof(buffer) - 1] = '\0';
```

应当严格检查边界条件,使用strncpy，snprintf等带长度检查的处理函数。

内存越界读取

尝试读取数组或动态分配的内存块之外的内存，可能导致读取到无效或敏感数据，也是不安全的行为。

案例1

```
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  for(int i=0;i<10;i++)
  {
      std::cout << arr[i] << std::endl;// error1
  }

  std::cout << arr[10] << std::endl; // error2
  return 0;
}
```

改为

```
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    //安全的for循环
    for(int val:vec) {
        std::cout << val << std::endl;
    }
    //边界检查
    if(vec.size() < 10){
       return 0;
    }
    try {
      std::cout << vec.at(10) << std::endl;
    }
    catch(const std::exception& e) {
      //...
    }
    return 0;
}
```

```
  var arr [5]int = [5]int{1, 2, 3, 4, 5}
  for i := range arr{
    fmt.Println(arr[i])
  }
```

案例2

```
int main() {
    char charArray[] = {'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'};
    std::string str(charArray);
    std::cout << "Converted string: " << str << std::endl;
    return 0;
}
```

改为

```
   char charArray[] = {'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd', '\0'};
   std::string str(charArray);
```

或者

```
    char charArray[] = {'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'};
    std::string str(std::begin(charArray), std::end(charArray));
```

解决方案:

- 使用标准库容器（如std::vector,std::string）代替原始数组，它们提供边界检查；用范围for循环代替索引。
- 对数组结构操作时，严格检查边界条件。

内存泄漏

当程序分配了内存但未能在不再需要时释放它，导致这部分内存无法被再次使用，直至程序结束。长时间运行的程序可能会因此耗尽系统资源。忘记释放用new分配的内存是最常见的原因。

案例1

```
A* a = new A();
//...
a = new A();
delete a;
```

改为

```
  {
    std::unique_ptr<A> a(new A());
    //a在离开作用域时会被自动删除
  }
```

案例2

未回收资源

```
func worker() {
  for {
    fmt.Println("Worker is running...")
    time.Sleep(1 * time.Second)
    }
}

func main() {
  for i := 0; i < 1000; i++ {
    go worker()
    }
    ...
}
```

改为

```go
func worker(wg *sync.WaitGroup, done chan struct{}) {
	defer wg.Done()
	for {
		select {
		case <-done:
			fmt.Println("Worker is stopping...")
			return
		default:
			fmt.Println("Worker is running...")
			time.Sleep(1 * time.Second)
		}
	}
}

func main() {
	var wg sync.WaitGroup
	done := make(chan struct{})
	for i := 0; i < 1000; i++ {
		wg.Add(1)
		go worker(&wg, done)
	}
	time.Sleep(10 * time.Second)
	close(done)

	wg.Wait()
}
```

案例3

异常处理不对

```
class MemoryManager {
public:
    MemoryManager(size_t size) {
        data_ = new int[size]; // 动态分配内存
        if (size >= 1024) {
            throw std::runtime_error("size too big");
        }
    }
    ~MemoryManager() {
        delete[] data_;
    }
private:
    int* data_;
};

int main() {
    try {
        MemoryManager manager(2000);
    } catch (const std::runtime_error& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
    return 0;
}
```

改为

```
class MemoryManager {
public:
    MemoryManager(size_t size) {
        if (size >= 1024) {
            throw std::runtime_error("size too big");
        }
        data_ = std::make_unique<int[]>(size);
    }
private:
    std::unique_ptr<int[]> data_;
};
```

悬挂指针（Dangling Pointer)

当一个指针指向的内存被释放或重新分配后，该指针就成为了悬挂指针。继续使用悬挂指针同样会导致未定义行为。这是一种**相当常见**的问题。

案例1

```
A* a = new A();
//1
delete a;
//*a，error
```

实际应用中，代码的情况会更加五花八门，更复杂一些，如下一些情况：

案例2

函数返回局部变量指针

```
A* new_a(){
  A a;
  //init...
  return &a; //error
}
```

案例3

B的实例b不由A管理，A难以确定其生命周期，甚至可能已经被重新分配

```
class A
{
public:
  A(B* b):b_(b){}
  ~A(){}
public:
  B* b_;
}

B* b = new B();
A a(b);

delete b;
//*a.b_
```

案例4

指针传入其他函数被修改，调用方不知道

```
void change_a(A* a)
{
  delete a;
}

A* a = new A();
change_a(a);
//*a error
```

案例5

隐蔽的变更，如vector扩容

```
std::vector<int> vec = {1, 2, 3};
int* elem = &vec[0];
//vec.push_back()超过capacity
vec.resize(vec.size() * 2);
//*elem,error
```

案例6

异步回调，指针已经被释放

```
A* a=new A();
auto f = std::bind(doSomething, a);
std::future<void> fut = std::async(std::launch::async, f);

delete a;
```

解决方案:

- 使用RAII（Resource Acquisition Is Initialization）原则管理资源。
- 使用智能指针(如std::unique_ptr, std::shared_ptr)，自动管理内存生命周期。

RAII理念的探讨

C++编程中的一个核心理念，全称为“资源获取即初始化”。这一编程范式通过将资源的管理与对象的生命周期绑定在一起，来自动管理资源（如内存、文件句柄、网络连接、锁等），确保资源在不再需要时能够被正确且及时地释放，即使遇到异常情况也是如此。

RAII的基本原则:

- 资源获取：在对象构造时获取资源（如分配内存、打开文件）。
- 资源释放：在对象销毁时自动释放资源（如释放内存、关闭文件），这通常通过对象的析构函数实现。

RAII通常通过以下几种方式实现：

- 自定义类来管理资源：设计一个类，其构造函数负责获取资源，析构函数负责释放资源。例如，自定义一个文件管理类，在构造函数中打开文件，在析构函数中关闭文件。
- 使用标准库容器和智能指针：如std::vector、std::string、std::unique_ptr、std::shared_ptr等，它们自动管理内存，无需手动释放。

在C++编程中，实现RAII，需要考虑所有权和生命周期问题。作为参考，Rust所有权系统的要求如下：

- 所有权唯一：任何给定时间，数据只能有一个所有者。
- 值的移动：当值从一个地方转移到另一个地方时（例如，作为函数参数传递或赋值给另一个变量），所有权也随之转移，而不是复制数据。
- 资源管理：当所有者离开作用域时，自动清理所占用的资源。
- 借用：允许其他部分的代码通过获得引用的方式临时访问数据，而不转移所有权。这要求同时只存在多个不可变的借用，或一个可变借用，并且借用者的生命周期应该小于或等于所有者的生命周期。

在C++中，前三项可以通过unique_ptr近似实现，而共享内存问题则只能靠开发者自己去管理，或使用shared_ptr，通过增加引用计数的方式解决。

智能指针的循环引用问题

智能指针是一个实现RAII原则的通用解法，但是同样存在一些问题。
除了性能损失，最容易发生的问题是让资源的管理复杂化，容易引发循环引用的问题，这可能导致内存泄漏。

解决方案
使用std::weak_ptr来替代直接的std::shared_ptr互相持有，从而打破循环引用。

案例1

```cpp
class A {
public:
    A(std::shared_ptr<B> b) : b_(b) {}
    std::shared_ptr<B> getB() const {
        return b_;
    }

private:
    std::shared_ptr<B> b_;
};

class B {
public:
    B(std::shared_ptr<A> a) : a_(a) {}
    std::shared_ptr<A> getA() const {
        return a_;
    }

private:
    std::shared_ptr<A> a_;
};
```

改为：

```
class A {
public:
    A(std::shared_ptr<B> b) : b_(b) {}
    std::shared_ptr<B> getB() const {
        return b_.lock(); // 转换为 std::shared_ptr
    }

private:
    std::weak_ptr<B> b_;
};

class B {
public:
    B(std::shared_ptr<A> a) : a_(a) {}
    std::shared_ptr<A> getA() const {
        return a_.lock(); // 转换为 std::shared_ptr
    }

private:
    std::weak_ptr<A> a_;
};
```

内存问题排查处理

c++

动态检查:在程序运行时监控内存的分配、使用和释放，能够检测到实际运行过程中的内存错误。这类工具通常会对程序性能有一定影响，但能提供详细的错误报告。

- **Valgrind** 一个强大的内存调试工具，它提供了多种工具，其中Memcheck是最常用的，用于检测内存泄漏、未初始化的内存访问、越界访问等问题。
  valgrind --leak-check=full ./your_program
- **AddressSanitizer** 集成在Clang和GCC中的一个快速的内存错误检测器，支持检测各种内存错误，包括缓冲区溢出、使用未初始化的内存、释放后使用等
  g++ -fsanitize=address your_file.cpp -o your_program
- 如果用到了如TCMalloc等内存分配器，可以查看堆内存分配情况来查找问题。也可以自行重载new delete，额外输出一些信息来进行排查。

静态检查:在编译时分析代码，不实际运行程序，可以发现潜在的编程错误和不良编程习惯，但可能无法捕捉到运行时才出现的动态问题。

- **Cppcheck** 一个静态代码分析工具，可以检测出未使用的变量、内存泄漏提示、数组越界等潜在问题。
- **Clang Static Analyzer** 是LLVM项目的一部分，它可以检测出一系列编程错误，包括内存泄漏、逻辑错误等。
- 基于**AI大模型**的编码辅助工具。
- **Code Review**。

golang

- **pprof** Go 标准库中的 pprof 包提供了一套强大的性能分析工具，可以帮助你收集 CPU 和内存使用情况的统计数据。