360漏洞研究院
VULNERABILITY RESEARCH INSTITUTE
清華大學计算机科学与技术系
Department of Computer Science and Technology, Tsinghua University

智能体安全
实践报告

目录

一、 概述

随着 LLM 推理预测能力不断提升进步，人工智能技术正进入积极探索应用落地的高速发展时期。作为生成式 AI 的核心交互方案，智能体（Agent）由于其能够进行环境感知、自主决策、任务执行的高度智能化特性，市场规模和应用场景持续扩大，展现出多样化发展趋势。与此同时，AI Agent 带来的安全风险也与日俱增，作为需要独立完成复杂任务的计算机工程应用，在面对不可信的网络环境和潜在攻击威胁时，如何确保其正确性和可靠性尤为重要。

近期，360 漏洞研究院联合清华大学计算机科学与技术系，针对 AI Agent 生命周期链路中的各个场景开展安全研究。通过典型攻击面梳理和漏洞挖掘，深入分析探讨了其中潜在的安全风险。结合 360 安全智能体的高效代码分析能力以及特有的特征库，研究团队报告了智能体相关开源项目漏洞 20 余个，并在此基础上对重点攻击场景进行归纳总结。

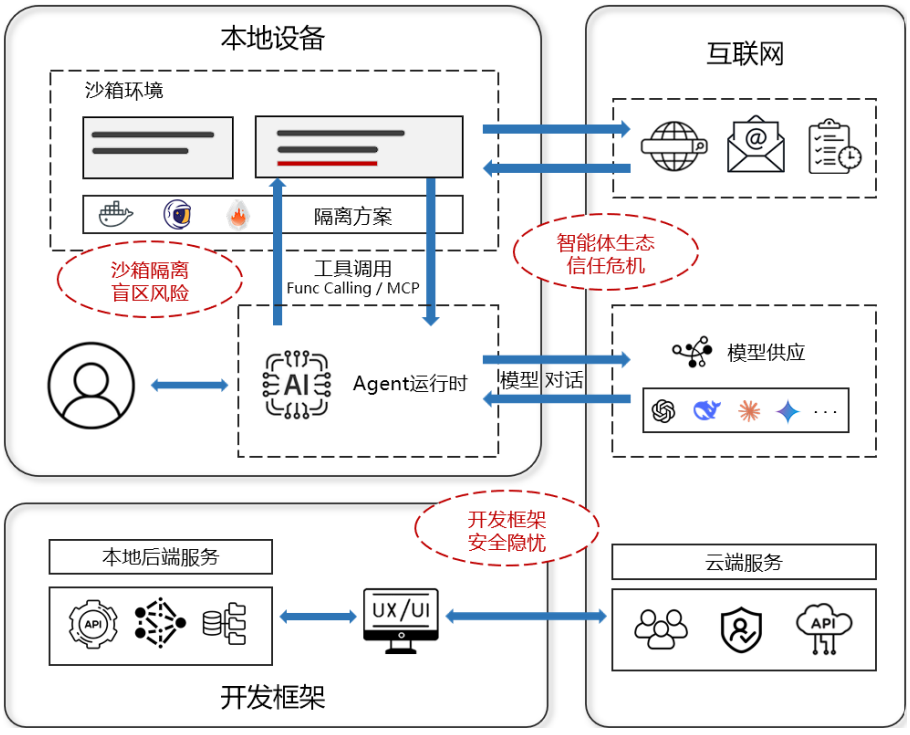

图 1-1 智能体安全实践场景概览

研究发现，AI Agent 的全生命周期安全风险呈现多维性、隐蔽性与系统性特征，其安全威胁渗透到开发、测试、部署和运营等一系列流程之中。本报告将 Agent 安全实践划分为框架层开发设计、生态层协同交互、沙箱层边界隔离三大具体场

景，详细介绍了场景中的运行模式，并对其中代表性的风险面和漏洞进行举例说明，旨在提供智能体安全的综合性视角，为智能体安全生态的持续、积极发展贡献力量。

二、 漏洞列表

目标名称	漏洞描述	CVE 编号
LangChain-Chatchat	任意文件写	CVE-2025-6853
LangChain-Chatchat	任意文件读	CVE-2025-6854
LangChain-Chatchat	任意文件写	CVE-2025-6855
n8n	命令注入	暂无
Flowise	参数校验错误	暂无
DB-GPT	参数校验错误	CVE-2025-6772
SuperAGI	任意文件写	CVE-2025-6280
Agent-Zero	任意文件读	CVE-2025-6166
AstrBot	任意文件读	CVE-2025-48957
Intel OpenVINO	信息泄露	CVE-2025-22892
XAgent	参数校验错误	CVE-2025-6281
Upsonic	远程代码执行	CVE-2025-6278
Upsonic	任意文件写	CVE-2025-6279
PySpur	远程代码执行	CVE-2025-6518
Steel Browser	任意文件写	CVE-2025-6152
OpenAgents	任意文件写	CVE-2025-6282
Sim	任意文件读	CVE-2025-7107
Sim	权限配置错误	CVE-2025-7114
Rowboat	权限配置错误	CVE-2025-7115
Xata Agent	任意文件读	CVE-2025-6283
Python-a2a	参数校验错误	CVE-2025-6167

在后续章节的分析中，使用了上述漏洞的部分详情信息作为实际案例，此外，

案例分析还包含了智能体设计中的风险因素，该类因素可能并不构成完整可用的安全漏洞，但仍可用于揭示当前智能体开源软件项目中易受攻击的不安全场景。

三、 开发框架中的安全隐忧

Agent 架构通常由模型（Model）、工具（Tools）、编排（Orchestration）三个主要组件构成。模型作为系统的决策核心，能够根据具体的输入指令进行推理和预测；工具提供了与外部数据和服务交互的接口，极大的扩展了 Agent 实时获取信息和处理复杂任务的能力；编排则决定了系统如何拆分任务，并根据推理结果来指导和规划之后的行动，直至完成既定目标。

Agent 开发框架通过对系统组件进行一定程度的抽象表达，以模块化、可扩展性和快速编排能力为核心，提供了一系列预设工具和基础功能，旨在简化智能体构建与部署流程，提升整体开发效率。然而，在框架为开发者带来便利性的同时，框架中的潜在安全问题也提供了额外的攻击向量，使其变为恶意攻击者通过网络发起攻击的“帮凶”。

1. 本地请求攻击

区别于那些暴露在广域网上，接受任意外部连接的公共服务，本地服务通常无法通过公网路由直接进行访问，并且预期只会收到局域网或者仅来自本地的访问请求。例如，当把服务配置为监听 0.0.0.0 这一特殊地址时，意味着其它进程可以通过该机器上的任意网卡 IP 访问到该服务，包括运行在其中的虚拟机，以及局域网内可达的其它设备。而如果将服务配置为监听 127.0.0.1（localhost），则该服务只能收到本地请求。

对于 Agent 开发框架而言，启动本地服务是最常见的行为之一，因为框架通常需要提供后端服务，来帮助开发者进行应用的调试和管理，而这类服务一般也仅需处理来自局域网或本地的请求，而无需运行在广域网中。正因如此，框架服务通常默认所有的请求都是可信任的，并且缺少对请求发起方的身份验证以及对请求中包含数据的二次检查。因此，对于绑定 0.0.0.0 地址的框架服务，可能成为攻击者进行横向渗透，或者容器逃逸的目标。

此外，即使服务限制 localhost 作为监听地址，仍有可能通过外部网络向本地服务发送请求，并利用其中的安全漏洞完成远程攻击。最为典型的场景就是将浏览器作为跳板，与浏览器所在设备上的本地服务建立连接，其流程如下图所示。

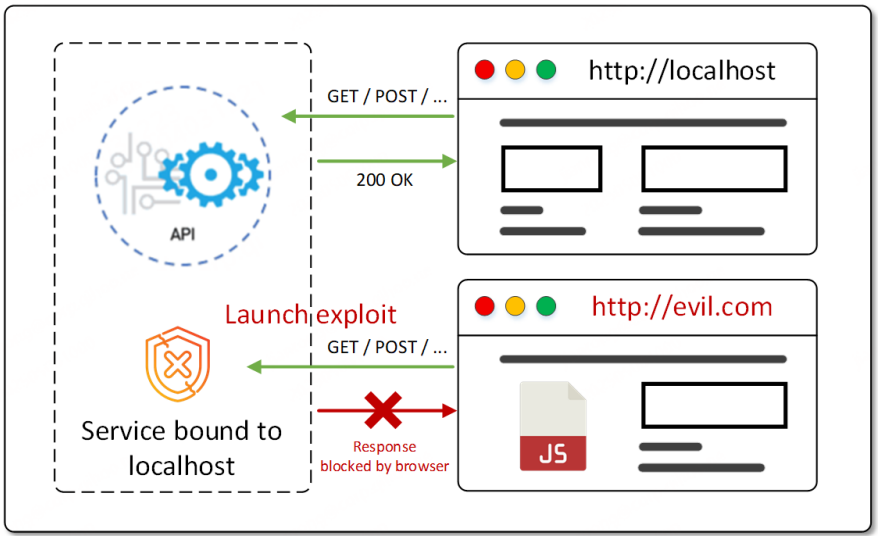

图 3-1 通过浏览器发起本地请求攻击

由于同源策略的限制，当从浏览器中发起跨域 HTTP 请求时，如果响应报文中没有包含正确的 CORS 响应头信息，则浏览器将阻止发起方获得响应内容。但上述攻击仍能成功的关键因素在于：浏览器虽然阻止了发起方接收响应，但并不会阻止发送请求这一行为，而有时候，单纯的发送请求就足以完成漏洞利用。

案例 1：Pyspur

Pyspur 是一个轻量级 Agent 工作流构建框架，支持以可视化节点编排的方式来生成复杂流程，每个节点代表了诸如 LLM 对话、工具调用、逻辑控制等特定功能。

在某些节点中，由于存在对 Jinja2 库的 Template 对象不安全的使用，因此可导致任意代码执行。Jinja2 是使用 Python 实现的一个模板引擎，支持在模板渲染过程中执行其中的表达式，服务端在没有良好过滤的情况下接收外部传递的数据并进行渲染，则可能会执行攻击者控制的恶意指令。

Pyspur 则是一个符合本地请求攻击场景的典型例子：将服务绑定于 localhost，且没有身份认证措施，意味着默认信任来自本地的请求；同时，触发其中的模板注入漏洞流程，仅需连续的发送几个请求，而不要求能够获取响应内容，意味着通过浏览器发起跨域请求能够满足攻击条件。因此，攻击者可以借助该场景轻易

实施针对该框架的远程代码执行，从而控制开发者的设备。

案例 2：Google Agent Development Kit

Agent Development Kit（ADK）是由 Google 发布的一个用于 Agent 工作流开发的工具集，并专门针对 Gemini 生态进行了优化，同时兼容其他模型与框架。ADK 提供基于 FastAPI 的本地服务，帮助开发者以命令行或 Web 可视化的形式来运行、调试、评估 Agent。

在启动时，服务将绑定于 0.0.0.0 地址，且绑定过程没有提供修改地址的选项，而是直接硬编码在代码中。这代表除非开发者自行寻找并修改软件源代码，否则无法将服务连接的访问限制在本地，同时，ADK 官方的文档中并没有体现这一设计特点，反而在某种程度上表明服务的连接仅需在本地完成。

图 3-2　ADK 官方文档说明

除此之外，ADK 的服务中还增加了对于 WebSocket 连接的支持，开发者可通过*/run_live* 接口与 Agent 建立实时性更强的连接，并通过 WebSocket 进行双向通信。这进一步削弱了浏览器对于跨域请求的防护，因为在默认配置下，浏览器不会阻止跨域建立 WebSocket 连接，也不会阻止发起跨域连接一方获得响应消息。因此，攻击者可以通过构造一个恶意页面，静默的与运行在 ADK 服务下的 Agent 建立通信并使用其实现的各项功能，从而造成更大的破坏。

2. 云上服务接口

框架中包含的软件基础设施通常提供多种部署方式，开发者可将其部署在自己的本地环境，或者以付费的方式直接使用框架部署在云端的服务。这也是许多开源框架的商业化模式之一。相较于本地部署，云端服务会增加账户认证和环境隔离等措施，用以确保用户的基本安全，但服务本身的核心代码逻辑与开源部分保持一致。

这种云服务的形式将一部分用户本地资源转移至云端处理，从安全的角度来看，原本由用户自行管理的本地资源，现在需要依靠服务提供者来提供安全保障。这也意味着如果云服务接口中存在漏洞，则攻击者就能利用它来影响到整个业务系统的安全。

案例：Steel Browser

Steel Browser 是一个为 Agent 提供浏览器操作接口的基础设施开源框架。它提供了一个完整的浏览器实例，允许用户使用编程的方式实现会话管理、页面提取、JavaScript 代码执行等自动化操作，其底层原理是通过远程调试协议与浏览器实例建立连接，并对外提供包装后的 REST API 服务，将各接口的功能转换为调试协议的请求，从而对浏览器实例进行控制。Steel Browser 有商业化的云部署，并提供了浏览器凭证管理和反机器人检测等功能。

在 Steel Browser 提供的 API 中存在文件上传功能，普通场景下，用户使用包含有文件内容的 POST 请求向服务端发送数据，并由服务端完成存储，此时会对请求中提供的文件名称进行检查，过滤其中的相对路径字符，以避免在后续路径拼接时出现路径穿越。然而，Steel Browser 的代码实现中，还支持了 fileUrl 字段，用于从指定的 Url 下载文件。在这个情况下，代码直接通过正则匹配的方式获取文件名，而没有进行额外检查，因此产生了路径穿越漏洞。

```
1 const nameMatch = disposition.match(/filename="(.+)"/i);
2 if (nameMatch && nameMatch[1]) {
3   name = nameMatch[1];
4 } else {
5   name = url.split("/").pop() || "downloaded-file";
6 }
7 filePath = join(tmpdir(), `upload_${Date.now()}_${name}`);
8 const result = await saveWithChecksum(streamFromUrl.stream, filePath);
```

图 3-3 Steel Browser 路径穿越相关代码

该漏洞可允许攻击者在指定路径下创建内容可控的文件，通过写入系统定时任务、敏感配置文件等方式，可进一步获取任意代码执行的能力。

3. 小结

Agent 开发框架在致力于降低 AI 开发门槛，提供智能体系统构建一站式服

务的同时，也因自身缺少完善的安全体系设计而存在着安全隐忧。无论是仅建立在本地服务，或是部署在云端的接口，都存在着从远程被攻破的可能。通过浏览器攻击私有网络的问题已经存在了很长一段时间，又因为开发框架通常承载着丰厚的计算与存储资源，可能会成为攻击者优先选择的目标。Chrome 作为市场占有率最高的浏览器，提出了专用网络访问（Private Network Access，PNA）规范，用于阻止从公网访问私网资源，但却因为各种兼容性问题，在 2024 年底宣布推迟 PNA 在浏览器上的启用时间，可见本地请求攻击仍将持续存在。

因此，开发框架应尽可能将本地服务默认绑定 127.0.0.1 而非 0.0.0.0，避免局域网其它设备访问，同时在服务层增加访问控制和身份验证，以阻止通过浏览器发起的本地请求攻击。而通过云部署的方式来使用框架中的部分功能，则更应关注接口实现中的安全设计，避免因接口暴露所引发的各类安全问题。

四、 智能体生态中的信任危机

自 2023 年 OpenAI 发布 Function Calling 功能支持大模型通过结构化编程方式调用外部函数以来，Google、Meta 等知名模型开发方均开始相继支持大模型的工具调用，随着工具连接更多软硬件设施，基座推理能力不断增强，Agent 被赋予解决现实生活中问题的能力。同时，AutoGen、LangChain 等主流开发框架通过提供额外的抽象层，来标准化不同模型中的工具解析和调用流程，从而提供多 Agent 共同协作的功能架构。

随着业务复杂度的提升，参与到 Agent 系统运作中的成员也越来越复杂，模型调用方、模型供给方、工具实现方、资源提供方等多个角色，构成了整个系统运转的基础。正因如此，Agent 系统的整体安全性，需由每一个参与方共同保障。

1. 调用链风险互嵌

（1）大模型输出

在 Agent 系统中，大模型通常作为核心的感知与决策模块，负责解析用户输入、规划任务流程，并生成用于调用外部工具或服务的指令。在复杂的系统设计中，往往存在多个大模型来扮演不同的角色。大模型的输出结果在很大程度上决

定了 Agent 的行为走向，包括是否调用某个工具、如何处理外部数据以及与用户或其他 Agent 的交互策略。因此，攻击者可能通过构造诱导性 Prompt，操控大模型生成包含恶意内容或错误流程的响应，从而间接影响 Agent 的行为。例如，攻击者可以诱导模型返回伪造的函数调用请求、嵌入非法指令，或故意提供错误信息，使 Agent 在毫无察觉的情况下执行越权操作、暴露敏感数据或破坏既定流程。这类攻击方式通常不依赖底层权限或接口漏洞，而是通过影响模型输出内容干扰 Agent 的决策，具有高度的隐蔽性和通用性。

目前，大多数 Agent 系统对大模型返回结果缺乏有效的安全验证机制，往往默认将模型输出视为可信输入，直接驱动后续工具调用或任务执行，导致潜在风险难以及时识别与阻断。因此，必须将大模型输出结果作为系统安全防护的一部分，引入包括内容合规检测、危险指令过滤、上下文一致性检查等在内的安全验证机制，从而提升 Agent 面对复杂环境和诱导性攻击时的稳健性与可信度。

（2）工具调用

工具调用作为 AI Agent 与现实世界深度交互的重要桥梁，极大拓展了智能体的能力边界，使其能够通过连接外部计算资源、数据接口及物理执行终端，构建“感知-决策-行动”的完整闭环。这种机制通过将算法决策与实体工具链动态结合，既扩展了 AI 的认知维度，又增强了执行效力。

为了使得大模型能够借助工具执行任务，开发者需要先向大模型注册工具功能描述和接口定义等信息，然后对大模型工作时的输出进行解析，判断提取其中的工具调用需求，并以特定的参数形式调用指定工具接口，最后将工具运行结果返回至大模型。Function Calling 是目前最广泛采用的实现方式，它允许模型通过自然语言生成调用指令，直接触发预定义函数完成操作。然而，在缺乏权限控制与参数校验的情况下，该机制存在显著的安全风险。攻击者可通过构造诱导性 Prompt 引导模型调用高权限函数，进而实现未授权操作、访问敏感数据，甚至远程执行系统命令，造成信息泄露或系统失控。

此外，Function Calling 这一开发流程在跨模型适配、多工具集成及不同架构部署场景下，往往面临存在代码实现碎片化和兼容性问题。对此，Anthropic 在 2024 年底提出了 Model Context Protocol（MCP）规范，通过标准化工具发现、注

册、调用等流程，提供大模型与工具之间的通用通信框架，使开发者无需考虑底层对接细节，提升 Agent 系统可扩展性与维护效率。

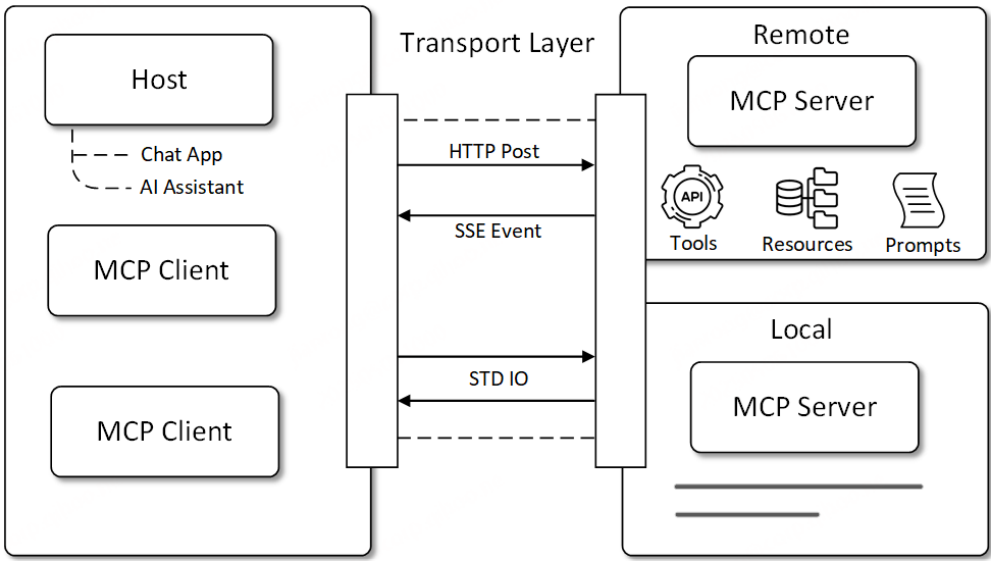

图 4-1 MCP 基本工作流程

Client-Server 架构进一步明晰了通信流程中各主体部分的职责：客户端负责与服务端建立并维护一对一的连接，使用特定的传输层协议进行数据通信；服务端实现具体功能，并有能力对外部的各类资源进行访问；主机应用可以根据实际需求对多个服务进行调用。

然而，在多方协同的场景下，包括 MCP 在内的主流方案并没有很好的解决系统中不同主体间的信任问题。在 MCP Server 井喷式增长，AI 应用接入工具愈发多样，复杂任务所需调用链条越来越长的背景下，该架构也显现出一系列的安全风险。

MCP Server 投毒。MCP Server 的组件包含有服务的元数据、配置信息，以及内部实现用于提供特定功能和资源的代码。Server 对于 Agent 系统能够产生控制流和数据流两方面的影响。使用 stdio 模式启动的 Server，通过标准输入输出管道来传递数据，需要运行在与 Client 相同的环境之中，其本质与直接运行互联网中下载的普通程序并无不同。由于缺乏统一安装渠道和严格的审计校验，Server 可以通过执行恶意代码的方式直接对系统产生严重的破坏。另一方面，经投毒后的 Server 可将恶意指令嵌入到工具的描述中，在影响模型的同时尽可能的隐藏指令内容，从而达到未经用户授权而执行操作的攻击效果。这类攻击可在 Server 生命周期的多个阶段发生，例如，当前主流 MCP 服务平台（如 mcp.so）

对用户上传的 MCP 服务缺乏审查，攻击者甚至可以上传名字和描述与其它已安装 Server 相似的名称和功能描述来实施伪装，并在注册工具时向接口描述中加入额外的恶意指令。

MCP Server 远程风险。在 sse（server-sent events）模式下，MCP 服务作为远程服务运行，与 Agent 通过持久化通信通道保持实时交互，同样会引入新的安全风险。首先，Agent 对远程 Server 缺乏运行时控制能力，攻击者可通过服务响应动态注入恶意数据流，诱导模型执行越权或错误操作。其次，sse 通信若未加密或缺乏身份校验，容易遭受中间人攻击，导致指令被监听、篡改或伪造。更严重的是，部分 Server 具备广播能力，若被控制，攻击者可利用其向多个智能体扩散恶意指令，形成跨智能体的投毒传播链条，进一步放大安全风险。

MCP Client 恶意请求。Server 中的工具实现通常需要接收来由 Client 传递的参数，并有能力执行代码或访问数据资源，因此在处理外部请求时，有可能出现一些传统的安全问题。例如，数据未经过严格的验证和过滤，可能导致 SQL 注入、跨站请求伪造等经典漏洞。同时，因 MCP 协议缺乏良好的身份验证和权限控制功能，导致私有 MCP Server 可能存在未授权访问风险。

（3）多智能体协同

所谓独木不成林，虽然 MCP 扩展了智能体处理任务能力的深度，但单一智能体的能力往往是有限的，当处理的任务跨越多个领域时，往往得不到正确的结果。因此，多个不同功能智能体之间的协同，打破信息孤岛，可以为智能体生态注入更强大的活力。

为了解决智能体之间如何进行有效地沟通协作问题，Google 提出了 Agent2Agent（A2A）协议，为智能体系统提供了一种标准化的交互方式，其核心能力如下：

1. 协议架构同样基于 Client-Server 架构，由 Client Agent 发起任务，并调度至不同的 Remote Agent 来执行；
2. 规定了 Agent 服务发现流程，使用 AgentCard 描述其身份、认证和能力等信息；
3. 规定了通信使用的结构体详情，用于支持任务周期、多格式数据交互、

状态通知推送等协议功能。

A2A 的提出与 MCP 相辅相成。当涉及到专业领域的任务时，可以通过将任务调度到不同智能体进行专业处理，每个智能体利用特定领域的提示词、知识库，同是灵活的通过 MCP 指导调用相关外部工具，对自身专业能力进行延拓。无独有偶，在 MCP 中常见的攻击向量，A2A 中也同样存在。

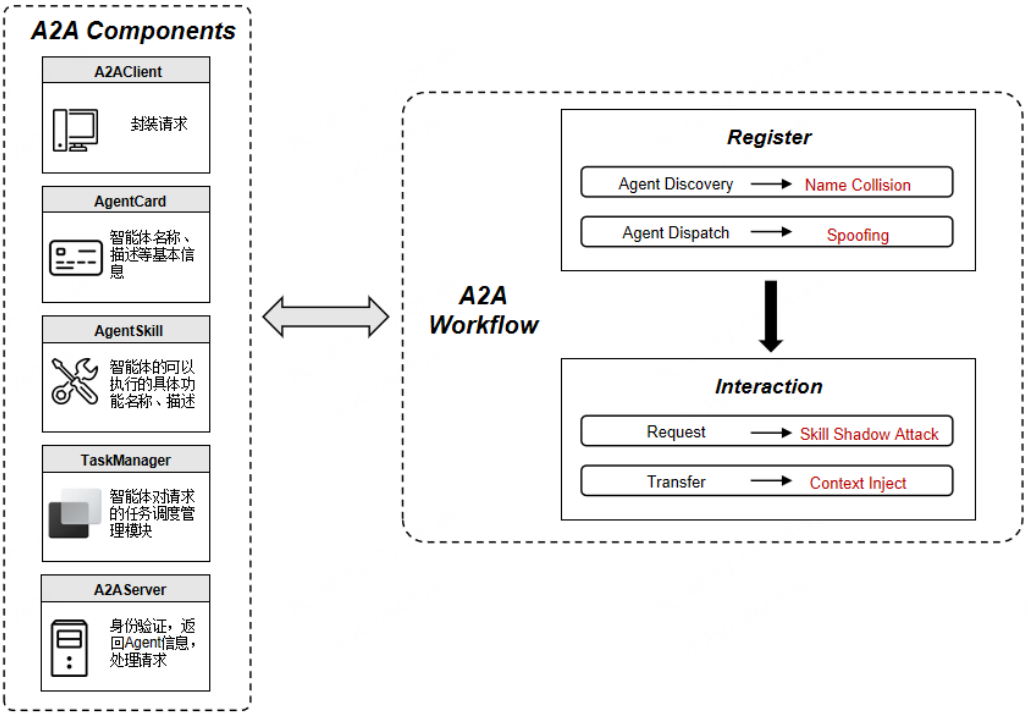

图 4-2 A2A 常见攻击向量

安全与认证。A2A 协议可借助 AgentCard 中的 authentication 相关字段指定身份认证形式，从而指导 Client 端在认证后对 Server 进行访问。但目前开源 A2A 协议实现并不包含具体认证接口代码，相反，协议建议开发者根据业务完成认证流程，并在后续交互中携带凭证与 Server 端通信。认证过程发生在 HTTPS 协议层，而与 A2A 协议消息无关，因此认证的安全性本质上取决于开发者自身的实现。

上下文投毒。多智能体系统一个重要功能在于打破信息孤岛，因此，一个智能体的输出可能会是另一个智能体的输入，如果一个恶意智能体返回的结果带有恶意注入内容，那么很有可能会影响下一个智能体的行为。

影子攻击。在 A2A 中，通过 AgentCard 描述智能体的基本信息，主要用于服务发现。AgentCard 中的 name 和 description 字段是 Client Agent 进行任务调度

的重要依据。例如，在官方的例子中，description 中的内容是提示词的关键部分：

```python
def root_instruction(self, context: ReadonlyContext) → str:
    current_agent = self.check_state(context)
    return f"""
    Please rely on tools to address the request, ...
    Agents:
    {self.agents}
    Current agent: {current_agent['active_agent']}
    """
```

图 4-3 A2A sample host agent 提示词

此外，AgentCard 中还有其它可被恶意利用的字段。例如，根据目前已有的一些客户端实现，展示给用户的一般是总体描述信息，而 AgentSkill 这类用于智能体进行确切请求的描述是隐藏的，向 AgentSkill 插入恶意提示就可能会在用户无法察觉的情况下，影响后续智能体的执行逻辑。

2. 脆弱的决策者

Agent 通过循环迭代的方式进行信息处理、知情决策，并基于前序输出来优化下一步行为，这一过程体现出了其实现目标的特定认知架构。该架构的关键组件是编排层，其承担着记忆维护、状态管理、推理运算与任务规划四项重要职能。编排层运用提示词工程技术及相关框架来指导推理与规划过程，通过将自然语言指令转化为结构化操作序列，使智能体能够有效感知环境特征并执行复杂任务。

从链式思维提示（Chain of Thought）到 ReAct 模式，Agent 的正确推理和响应很大程度上依赖于可靠的输入信息，但正如互联网上纷繁的信息经常使人难以判断真伪一样，对外部数据输入的强依赖性使得其易暴露于潜在的安全威胁之下：当认知层接收的数据被注入对抗样本或逻辑误导信息时，其决策可能产生极大偏差，通过设计特殊的数据污染，从而将 Agent 的自主决策导向预设的恶意逻辑之中。目前 Agent 设计的特性，使得这类诱导攻击实施起来难度不大，却可以产生严重的安全影响：

1. 决策核心需要观察工具执行后的结果来规划下一步骤，存在被恶意输入干扰的可能；

2. 工具往往需要访问和处理外部的不可信资源，例如抓取搜索引擎结果，或者工具本身即隐藏有恶意行为；
3. 决策核心观察过程中受到诱导，转而生成偏离预期的计划任务，即 Agent 行为完全由外部指令控制。

可见，产生这类问题的核心在于，负责决策的大模型在多步迭代的过程中无法准确区分原本用户的合理请求和源自攻击者的注入指令，从而会以较高的权限调度工具来执行恶意行为。

案例：Browser Use

Browser Use 融合了大模型编排和浏览器自动化工具，使得 Agent 能像真实用户一样访问网页并执行各项任务，例如网页导航、数据抓取和表单填写。该系统的大致工作流程如下：

1. Agent 从用户端接收任务描述，启动浏览器实例，完成系统初始化；
2. 将浏览器状态、历史记录、当前页面数据等信息与内置的 Prompt 进行结构化组合，并交由大模型决策；
3. 根据大模型的输出调用浏览器自动化接口执行所需动作，并将输出作为历史记录进行记忆存储；
4. 重复上述操作，直到 Agent 判断任务成功完成或无法继续。

可见 Browser Use 遵循了 ReAct 设计模式，通过观察浏览器的状态及页面信息来决定完成任务所需的步骤，然后调用工具来驱动浏览器依次执行，并循环该过程直至达到既定目标。

为了让决策模型理解网页内容并能够执行诸如填写账户密码和点击按钮等动作，Browser Use 会向网页注入 JavaScript 代码来构建 DOM 树信息，提取其中可交互的元素，组织为具有特定格式的文本，并根据模板拼接在 HumanMessage 结构体中发送至决策模型。而该文本中包含的对网页元素的描述并没有经过额外的过滤处理，因此形成了一个提示词注入场景，即来自外部互联网页面数据可直接拼在发送给决策模型的消息中，并对其后续推理产生极大影响。

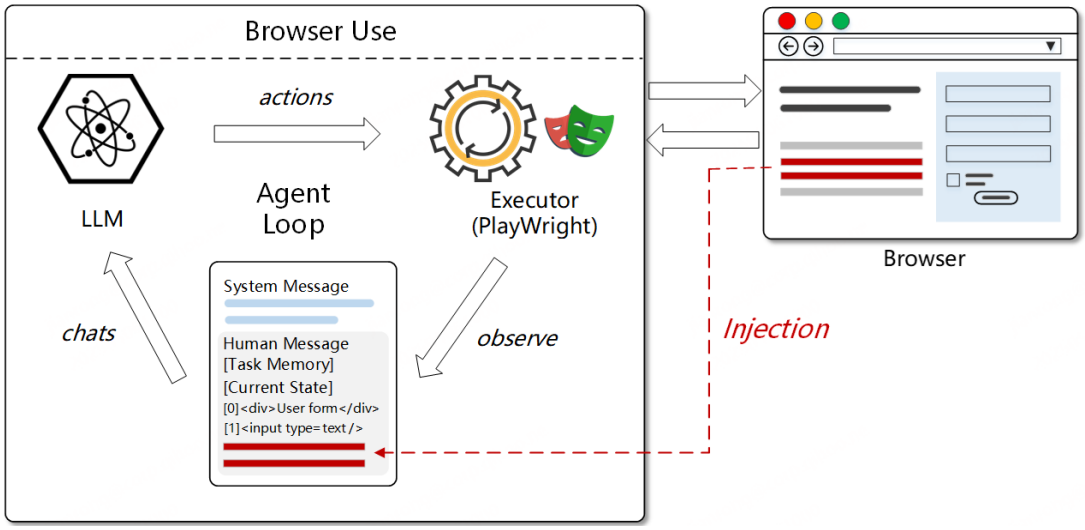

图 4-4 Browser Use 注入攻击

在实际测试中，我们发现通过注入攻击，可以对 Browser Use Agent 的任务流程实施接管，从而借助浏览器来执行攻击者指定的一系列行为，例如，利用 *file://* 协议加载设备上的敏感文件，并利用页面抓取功能将文件内容加入 Agent 记忆模块，最后导航至另一攻击者控制的网页，将记忆数据通过表单方式发送。此外，发起这类攻击时无需攻击者对浏览器访问的网页具有完整控制权，而是可以灵活选择邮件、评论、影响搜索引擎抓取结果等部分留痕的方式。

3. 小结

在 Agent 生态中，由于参与主体身份离散化、角色权责碎片化的特性，极易产生系统性安全风险。一方面，系统的正常运作依赖于各组件的协同合作；另一方面，调用链路中任何环节的恶意行为可以轻易打破信任，严重影响整个系统的安全性和可靠性。在单一的大模型对话生成场景下，攻击者经常使用数据污染或提示词越狱等方式，绕过模型的安全限制或过滤器，使得模型无法正常完成推理预测，但这类安全风险更多的集中在数据层面。而当大模型在工具调用的支持下能够真正观察并影响外部实体时，由于决策核心被误导而产生的破坏将被进一步放大，因此，目前大多数 Agent 还无法离开人的判断和参与。安全领域有一个经典论述：“人是网络安全中最薄弱的环节”，那么当未来 Agent 参与到生活的方方面面时，是否也会因为其易受到误导性数据的影响，而成为安全中的又一个薄弱环节？因此，在 Agent 的设计中应始终以不可信任视角来对待系统中不同的参与

主体，实现标准化的、安全的工具注册、发现和调用协议，以及针对所有实体的身份认证、授权和完整性检查。在易受到外部数据注入影响的 Agent 系统中，对处理这类数据的模型进行跟踪和约束，阻止模型的输出对下游构成风险的可能，通过合理的设计模式来避免即时注入型攻击带来的影响。

五、 沙箱隔离中的盲区风险

随着智能体可操作的工具种类愈发丰富，其能力边界也不断得到拓宽。从使用单一的代码解释器，到接入浏览器甚至完整的操作系统，智能体能根据实际需求来执行代码、查询网络、访问系统接口。然而，由于大模型输出内容具有一定程度的不确定性，其利用工具执行的操作可能是错误的，甚至是有害的，这使得智能体被赋予的过大自由度成为了一把“双刃剑”。为了避免这类操作对用户系统的整体安全性产生危害，通常会使用沙箱方案将工具执行操作的环境与真实的系统环境隔离开来，在沙箱内完成不安全的指令后，将执行结果以一种可观测的格式返回至沙箱外的大模型，从而确保工具使用的安全性。

通常来说，沙箱的防护效果主要取决于两个方面，其一是沙箱自身实现的完备性，需尽可能避免代码中存在可被利用的安全漏洞，导致攻击者能够完成沙箱逃逸；其二则是沙箱架构的合理性，需根据业务落地场景选择合适的沙箱，同时避免采用了不安全的配置而导致沙箱出现易被攻破的短板。本节对常用的沙箱进行梳理，并结合案例对不合理配置沙箱而产生的潜在风险进行分析。

1. 差异化沙箱选择

使用何种沙箱技术主要取决于 Agent 自身业务对安全性的要求，同时还需考虑运行沙箱所带来的额外性能开销。因此针对不同的场景，在沙箱的选择上也存在差异性。

代码执行层隔离。对于简单的计算任务，通常使用 Python 或 JavaScript 等解释执行类的编程语言即可完成需求。早期的一些框架产品提供了基于黑名单和代码扫描的沙箱实现，通过禁止代码调用特定的模块来过滤潜在恶意行为，但很快被证明无法提供预期的防护效果。因此，使用更为健全的权限控制运行时成为了

主流方案。以支持 TypeScript / JavaScript 代码运行的 Deno 为例，其沙箱隔离通过两个方面来实现：一方面，使用 V8 引擎作为代码执行器，在严格遵循 ECMAScript 准则的受限环境中运行，不具备文件读写、网络请求等能力；另一方面，向 V8 运行时中注册扩展函数，提供了外部资源访问的功能，同时引入权限管理模块来对所有涉及敏感操作的调用进行检查，从而确保代码在用户约束的权限下运行。

进程层隔离。进程层隔离方案使用操作系统中提供的各类安全机制来构建沙箱环境，其优势在于能在较低的性能开销下提供更加完善的运行基座和更为丰富的安全隔离特性。以 Docker 为代表的容器类沙箱，因其生态成熟、部署方便，逐渐成为 Agent 系统中不可或缺的一部分。Docker 使用 Namespace 来为不同的容器创建单独的资源隔离，使用 Cgroup 对容器能够使用的资源进行分配和管控，使用，同时借助 Seccomp、AppArmor 等内核安全功能来过滤系统调用，限制程序权限边界。Agent 可通过部署不同的 Docker 容器来完成工具与宿主机之间的隔离，以及工具与工具之间的隔离。

内核层隔离。由于 Docker 容器与宿主机共享同一系统内核，通过内核漏洞进行逃逸的风险始终存在，因此，在对生产环境安全要求较高的场景下，会采取内核层隔离措施。内核层隔离的重点在于减少沙箱内部环境对操作系统内核的依赖，从虚拟化的层面建立隔离带。例如，基于 KVM 的轻量级虚拟机管理 Firecracker，通过定制化裁剪内核中的一些驱动及子系统，实现可快速启动的 MicroVM。另一类则不使用虚拟机，将虚拟化边界从内核移动至系统调用，这类技术以 gVisor 为代表，通过对大量 Linux 系统调用进行沙箱化处理，实现了用户态内核，避免传统虚拟化设备导致的性能开销。

常用方案	防护层级	性能开销	应用场景
Deno	代码执行环境隔离	较小	运行无需依赖外部组件的解释执行类语言任务
Docker	进程层隔离	适中	信任边际内的应用隔离、资源控制和快速部署

Firecracker	硬件虚拟化内核隔离	较大	大规模、多租户下的计算场景服务基座
gVisor	用户态内核隔离	较大	隔离性要求较高场景下的不可信代码执行

主流沙箱方案对比

随着 Agent 对于沙箱需求的增长，也出现了许多“沙箱即服务”的云端隔离产品，例如 E2B 和 Daytona，用户可在无需关心沙箱部署的前提下，调用云服务来执行沙箱任务。

2. 易忽视的暗面

选择契合业务的沙箱方案是保障 Agent 系统安全的关键一步，但并不意味着就能高枕无忧。沙箱方案往往伴随着细粒度的安全配置选项，对于沙箱防护而言，其安全性具有明显的短板效应。由于面向智能体业务的沙箱往往依托于久经考验的成熟沙箱设计，开发者可能会下意识认为沙箱可以“开箱即用”，从而忽略了对沙箱进行安全配置的深入考量。

案例 1：OpenManus

OpenManus 是一款由 MetaGPT 团队开发，被设计为能够自主完成复杂任务的通用型智能体系统。开发者赋予了 OpenManus 较高自由度，其自带的工具集中提供 Python 解释器、命令行终端、浏览器等丰富的能力接口。目前 OpenManus 的许多工具并没有运行在沙箱环境中，但随着项目发展，开发者正逐步加入沙箱方案，将一些存在风险的操作进行隔离。

近期，OpenManus 针对其中用于查看、创建和编辑的自定义文件工具引入了沙箱配置支持。该设计通过启动一个 Docker 容器，在容器中设置用户的工作目录，然后将用户上传的文件置于沙箱内进行操作，从而在一定程度上实现了资源环境的边界隔离。

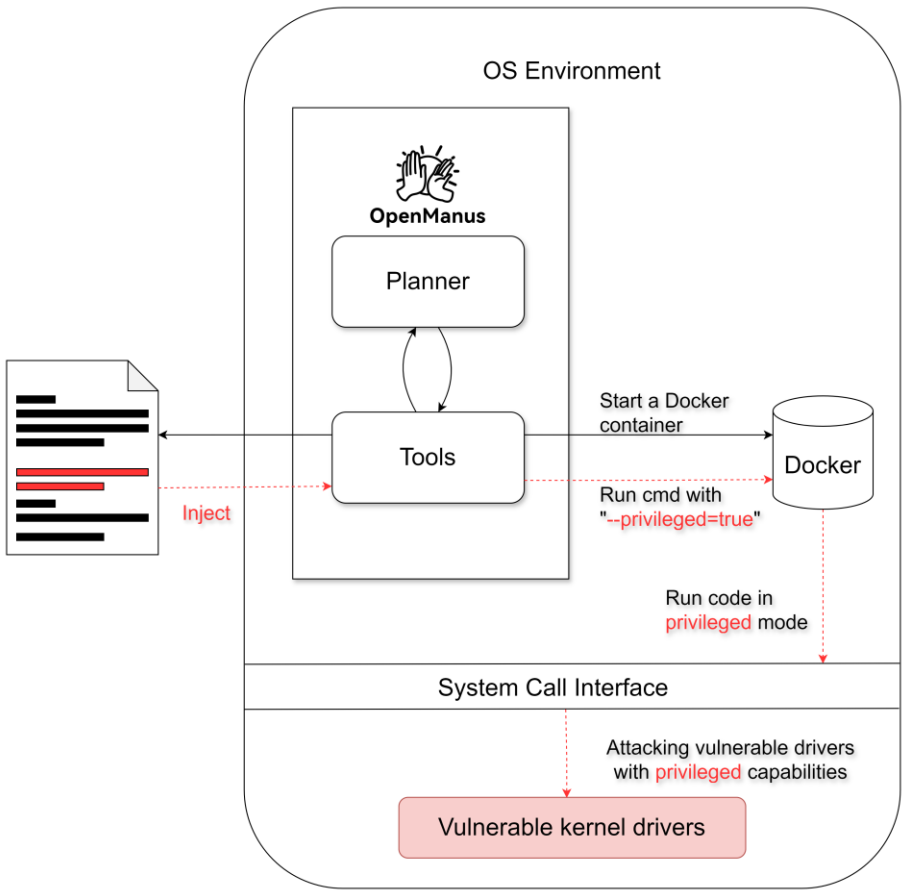

图 5-1 OpenManus 特权模式运行命令

通过分析发现，OpenManus 虽然在启动容器时使用了默认安全的参数配置，但却随即通过 docker exec 命令以特权模式在容器中建立了一个命令行终端，并借助该终端来执行命令。在特权模式下，进程被赋予了所有的内核 capabilities，意味着能够发起原本不被允许的特权内核调用，从而极大的拓宽了容器逃逸的攻击面。由此可见，一个关键参数的配置错误，可能导致整个沙箱环境的安全性受到影响。

案例 2：OpenHands

OpenHands 是一个基于人工智能的软件开发平台，旨在通过 AI 增强软件开发过程。该平台支持开发者执行各类任务，从代码编写到调试操作，乃至 Git 自动化版本管理。为了实现对目标软件开发环境的隔离，OpenHands 根据业务功能划分启动了两个 Docker 容器，一个用于运行与用户交互的可视化服务端，另一个则用于构建实际软件开发环境。经分析发现，该设计存在一定程度的安全风险：

1. 为了让服务端 Docker 能对开发端 Docker 进行管理，采用了 Docker In Docker 模式，将宿主机 docker.sock 套接字挂载至服务端 Docker 之中。

因此，服务端 Docker 本质上拥有了对整个 Docker 服务进行访问控制的权限，且进一步有了直接影响宿主机的能力；

2. 开发端 Docker 拥有完整的网络访问权限，且与服务端之间的通信不做限制，使得其能够直接使用服务端中的各类接口。

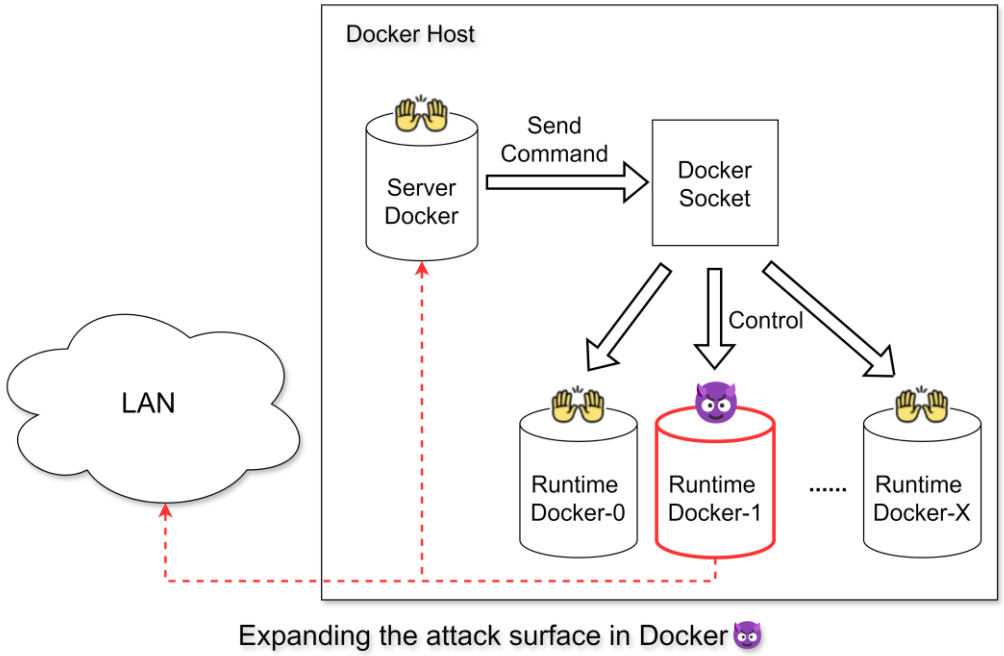

图 5-2 OpenHands Docker 配置中的风险

可见这种沙箱方案仅强调了利用 Docker 完成文件系统资源上的切分，但并没有严格安全设计上的考量。鉴于开发端需要运行大量的模型生成代码和测试，攻击者可在开发端 Docker 执行代码为起点的场景下，利用逐级跳跃提权的方式，完成沙箱逃逸。

3. 小结

在软件的安全开发中有一个重要概念：最小特权原则，即用户或实体只能被授予完成必要任务所需的特定权限。这对于 Agent 系统，尤其是通用型 Agent 系统而言也同样重要。Agent 应该仅接入其功能范围内所必要的工具，且需结合沙箱隔离方案来防止不当使用工具而导致的潜在安全问题。同时我们必须意识到，沙箱并非“银弹”，从文中分析的部分案例可见，当前开源应用的沙箱方案多聚焦于快速形成文件系统和代码执行的隔离环境，却忽略了智能体参与的具体场景下细化配置的重要性。未来的隔离机制需向多层级防护演进，以合适的沙箱技术

为基底，在数据层保障跨工具调用时的信息安全，在逻辑层强化模型异常检测，识别智能体偏离预设目标的行为模式，扫除隔离架构中的风险盲区，形成广义上的健全沙箱防护体系。

六、 总结

人工智能技术的快速发展正深刻重塑人类社会的生产与生活模式，具备自主决策与复杂任务执行能力 Agent，既带来了广阔的应用前景，也引发了前所未有的安全挑战。

在技术架构层面，开发框架的模块化与快速编排能力虽然显著降低了 Agent 的开发门槛，但其预设工具链与接口设计中的安全漏洞却可能成为攻击者的突破口。尤其是在当前 AI 浪潮下，越来越多的框架面向个人用户开发微型应用需求，强调便捷性的同时却牺牲了一定的安全性；在协作生态维度，智能体系统对多角色、多工具的整合能力使其安全边界愈发模糊，通过自然语言形式驱动智能体自主执行任务的特性天然使得系统易受到外界的干扰和影响，其核心矛盾体现在智能体的开放性协作需求与封闭性安全诉求的对立之上。

未来，随着智能体向工业控制、医疗诊断、金融决策等高风险领域渗透，其安全体系需同步进化。只有将安全性作为智能体技术演进的核心指标，而非事后补救的附加功能，才能推动智能体真正成为人类社会的可靠伙伴。

贡献者列表

360

龚　广　简　容　苏瑞泓　钟佳明　赵立伟　张　圆　潘剑锋

清华大学

徐　恪　李　琦　崔天宇　陈　淼　赵　乙