
OACAP Agents

Dmitrii Volkov¹
Palisade Research

With
Apart Research

Abstract

Building agents requires balancing containment and generality: for example, an agent with unconstrained `bash` access is general, but potentially unsafe, while an agent with few specialized narrow tools is safe, but limited.

We propose OACAP Agents, a framework for hierarchical containment. We adapt the well-studied paradigm of object capabilities to agent security to achieve cheap auditable resource control.

Keywords: Agent Security, AI Safety

1. Introduction

Building secure AI agents requires balancing flexibility with containment: highly capable agents with broad access pose safety risks, while narrowly constrained agents can be too limited. We propose OACAP Agents, a framework for hierarchical containment using object capabilities (OACAP). OACAP allows agents to control resources through a system of permissions that can be delegated, copied, or attenuated as tasks are split into sub-tasks. This ensures agents have only the necessary access for their specific task, while preserving the ability to manage complex operations safely and efficiently.

The framework supports recursive containment: an agent with broad permissions can spawn sub-agents with progressively narrower tasks and permissions, maintaining strict control over resource access. This approach contributes to the

¹ dmitrii@palisaderesearch.org

field of AI safety by offering a practical method for containing agents while allowing them to remain flexible and functional.

2. Overview

2.1 Sandbox

To control tooling access and reason about agent-tool interfaces, we first need to define an agent sandbox. We use a separation kernel design²: that is, an agent can only communicate with IPC messages; all IPC messages must be well-typed; IPC messages may be journaled in an audit log. An agent has no direct access to the system and requests tool access through IPC messages, which the security engine might refuse.

An agent might talk to another agent (provided it has its handle) via the same IPC. One way for an agent to get another agent's handle is to spawn a child by forking. This yields an IPC handle; no other resources are shared.

2.2 OCAP

Let's add resources, for example filesystem access, to the above system. To that end, we use object capabilities (OCAP for short). One can think of OCAPs as recursive versions of mobile OSes' permissions systems.

They work like so: a *capability* designates an object and permission to use it. A capability holder may *copy*, *delegate*, or *attenuate* a capability. Copying a capability yields two identical resource grants; delegating passes it to another agent and revokes the original's access; attenuating yields a weaker copy of the capability.

Here's a worked example:

1. Agent A has a `rwX` capability to `/home/A`.
2. It passes it to agent B and loses access to `/home/A`.
3. Agent B attenuates the capability to `rx` and passes it back to A. Now B has `rwX` access and A has `rx` to `/home/A`.

2.3 Agents

Now to grow some agents on this security substrate. Study of inner alignment and agent design popularized agent decomposition: the notion that a general-purpose agent might spawn sub-agents to tackle more narrow tasks. This connects beautifully to attenuating capabilities to get narrower resource grants.

Concretely, we propose the following design of recursive containment:

1. A top-level agent `A0` has full resource access `C0`³ and gets task `T0`
2. `A0` refines the task into subtask `T1` and attenuated capability `C1` and spawns subagent `A1(T1, C1)`

² Some readers might be familiar with the implementation in Qubes OS

³ Say, full filesystem access

3. (repeat 2 until tasks complete)

Worked example:

1. A0 has full access to user’s email and gets task “Send Inga an email asking to meet up tomorrow”
2. A0 finds Inga’s email and refines the task to {T1=“Send Inga <inga@example.com> an email asking to meet up tomorrow”, C1=“send to inga@example.com”}
3. A1 refines the task to {T2=“Write Inga an email asking to meet up tomorrow”, C2=∅}
4. A2 writes the email text and returns to A1
5. A1 sends the email with the C1 cap
6. A0 reports success to the user

Observe that each A_i has fewer permissions and a narrower task: A2, the agent writing the email text, has no capabilities at all.

This design generalizes to tree decomposition and more complex inter-agent communication; the example above is linear for simplicity.

3. Code

This is an idea-only submission.

4. Discussion and Conclusion

OCAPs are traditionally considered resource-intensive to implement: tracking attenuation and threading caps through an app requires careful abstraction design and adds some engineering overhead. Thankfully, agents can do this automatically.

One might object that securing an agent with an agent just passes the buck. We have two responses:

1. The “controller” agents refining the task and attenuating the capabilities and “worker” agents performing the tasks don’t have to be the same: it’s only the latter ones that have to be flexible in the containment—generality balance. We can plausibly contain controllers (they have a narrow job of only refining and attenuating), and keep workers general.
2. This design gives us a rich audit log of not only workers’ tool actions, but also controller decisions. This added transparency makes it easier to detect misalignment.

We hope this submission contributes to the field of knowledge of high-risk tooling control and agent-tool interfaces. OCAPs are a security engineering tool that seems especially suited to current agent design approaches; we describe a simple OCAP scheme to immediately contain one-shot agent applications, and sketch one for complex persistent applications.