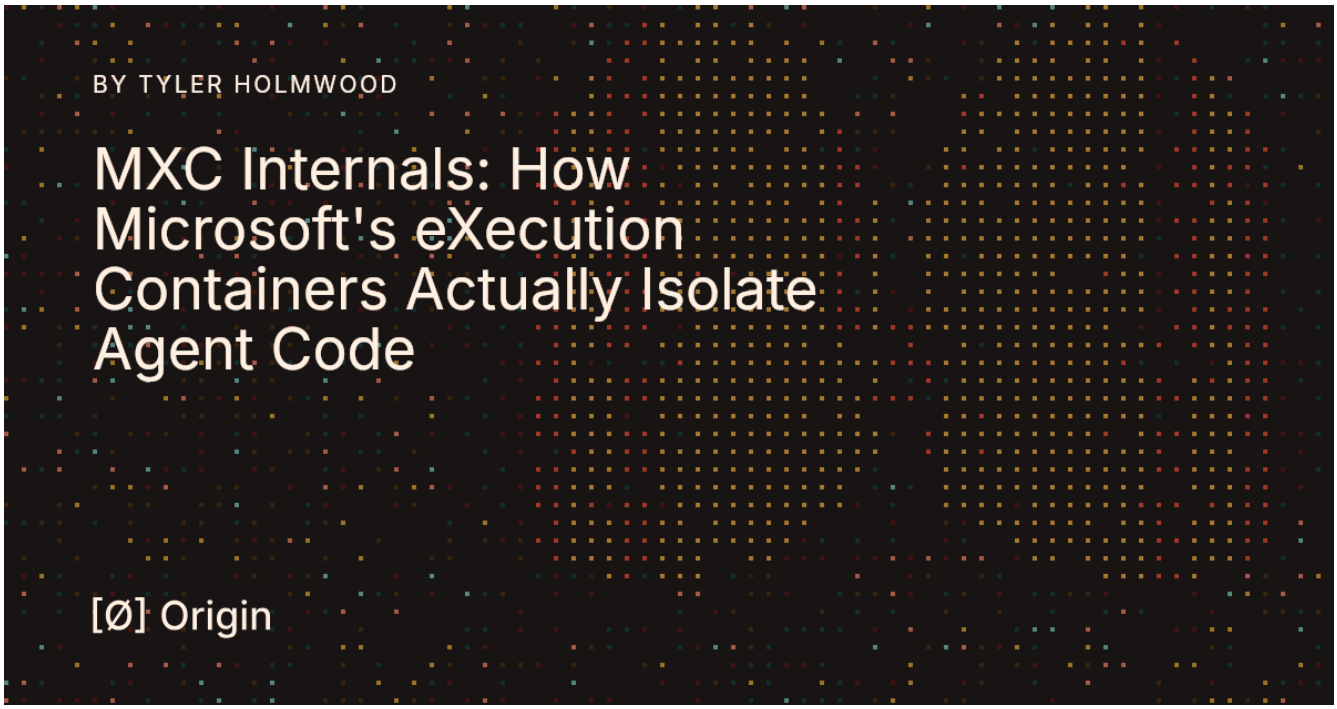
MXC Internals: How Microsoft's eXecution Containers Actually Isolate Agent Code

originhq.com/research/mxc-execution-containers-internals

Tyler Holmwood June 4, 2026



← Back to Research

Introduction

When an agent decides to run code, *where* does that code run, and what can it touch? Every coding-agent vendor now has an answer. OpenAI's Codex CLI sandboxes locally through OS-native primitives: macOS Seatbelt, Linux Landlock + seccomp; Anthropic's Claude Cowork runs the agent inside a full local Linux VM layered with seccomp and a network allowlist; hosted offerings like Google's GKE Agent Sandbox and LangSmith Sandboxes wrap the workload in a VM or container. But so far, no OS vendor has provided a native solution.

At Build 2026, Microsoft open-sourced MXC, the Microsoft eXecution Container, under the MIT license: "a sandboxed code execution system for running untrusted code (model output, plugins, tools) on Windows, Linux, and macOS."

We usually have to reverse engineer what we write about here, but this time Microsoft is putting it all out in the open. This is a tour of what the source does: one dispatcher binary, a JSON policy model, and ten containment backends. The backends get the most attention,

because that is where policy turns into real enforcement, from AppContainer capability SIDs and Job Object UI limits to Hyper-V micro-VMs and macOS Seatbelt profiles. There are also newly documented Windows API functions and some sneak-peeks into upcoming Preview build functionality.

Every excerpt below is copied from the repo and annotated with its file and line range.

> A maturity caveat. The README calls this an early preview and states plainly that "no MXC profiles should be treated as security boundaries currently." We document how the isolation is built, not whether it holds. Many of these backends sit behind `--experimental`, as noted in the summary table.

What MXC is

Strip the agentic framing and MXC is three things:

1. **A single native binary** `wxc-exec.exe` on Windows, `lxc-exec` on Linux, `mxc-exec-mac` on macOS, built from a Cargo workspace.
2. **A versioned JSON config** describing a command to run plus a policy (filesystem read-only / read-write / denied paths, network posture, UI restrictions).
3. **A TypeScript SDK** (`@microsoft/mxc-sdk`) that builds those configs and shells out to the binary.

The binary reads the config, picks a *containment backend*, translates the policy into that backend's native enforcement, runs the command, and streams back stdout, stderr, and the exit code. It is an *execution* layer, nothing more: pulling WSL images, warming Hyperlight snapshots, and enabling VM features are out-of-band steps you handle yourself.

A single `match` on the request's `containment` field selects the backend, in `core/wxc/src/main.rs`. The ten variants:

Wire name	OS primitive	Stability
`processcontainer` (default)	AppContainer **or** BaseContainer (runtime-selected, 3 tiers)	Stable (except BaseContainer)
`isolation_session`	`Windows.AI.IsolationSession` broker	Experimental
`windows_sandbox`	Windows Sandbox (Hyper-V disposable VM)	Experimental
`wslc`	WSL2 micro-VM + OCI container	Experimental

Wire name	OS primitive	Stability
`microvm`	NanVix micro-VM over WHP/KVM	Experimental
`hyperlight`	Hyperlight + Unikraft micro-VM, in-process	Experimental
`bubblewrap` (default on Linux)	`bwrap` user namespaces	Stable
`lxc`	LXC system containers	Stable
`seatbelt`	macOS Seatbelt (`sandbox_init`)	Experimental
`vm`	(not implemented)	Stub

Two execution models coexist. The **one-shot** model (`ScriptRunner` trait) pays full provision-execute-teardown on every invocation; every backend implements it. The **state-aware** model (`StatefulSandboxBackend` trait) exposes five lifecycle phases (`provision`, `start`, `exec`, `stop`, `deprovision`) for long-lived sessions. Only Isolation Session implements it today.

The rest of the post is the backends, one at a time.

ProcessContainer: AppContainer and BaseContainer

This is the default backend and the only stable one on Windows, so it earns the most attention. The wire value `processcontainer` resolves at runtime to one of **three isolation tiers**, picked by a fallback detector based on host OS support:

Tier	Primitive	Filesystem enforcement
1: BaseContainer	`Experimental_CreateProcessInSandbox` (processmodel.dll)	Native OS sandbox API
2: AppContainer + BFS	AppContainer + `bfscfg.exe` policy	BFS allow-lists
3: AppContainer + DACL	AppContainer + host NTFS ACEs	DACL grant/deny ACEs on host paths

Selection prefers the strongest tier the host can satisfy (fallback_detector.rs:131-252): Tier 1 if `processmodel.dll!Experimental_CreateProcessInSandbox` resolves; else Tier 2 if built with the `tier2_bfs` feature and `bfscfg.exe` is found; else Tier 3.

Tiers 2 and 3 both build on classic AppContainer process creation, so we start there; it is the clearest view of policy becoming kernel-enforced restrictions. Tier 1, which hands the whole job to the OS in one call, comes last.

Minting the identity

Everything starts with a security identity. The backend calls `CreateAppContainerProfile` to register a per-name profile and return its `S-1-15-2-...` package SID; if the profile already exists, it falls back to `DeriveAppContainerSidFromAppContainerName` to re-derive the same SID (appcontainer_runner.rs:384-416). So the same container name resolves to a stable principal across runs. That SID anchors everything downstream: it is what firewall rules and, in Tier 3, filesystem ACEs target.

Capabilities and SECURITY_CAPABILITIES

AppContainers start with essentially no access; capabilities grant specific classes of it. MXC takes the policy's declared capabilities, always appends a custom `AgenticAppContainer`, and adds `internetClient` when network access is allowed in capability mode:

```
// --- Build capability list ---
let mut capabilities_to_add: Vec<String> = request.policy.capabilities.clone();
capabilities_to_add.push("AgenticAppContainer".to_string());

let use_capabilities_for_network = matches!(
    request.policy.network_enforcement_mode,
    NetworkEnforcementMode::Capabilities | NetworkEnforcementMode::Both
);
if use_capabilities_for_network
    && request.policy.default_network_policy == NetworkPolicy::Allow
    && !capabilities_to_add.iter().any(|c| c == "internetClient")
{
    capabilities_to_add.push("internetClient".to_string());
}
```

(appcontainer_runner.rs:444-491)

Each name becomes a capability SID via Win32 `DeriveCapabilitySidsFromName`, is marked `SE_GROUP_ENABLED`, and is collected into a hand-rolled `#[repr(C)]` mirror of `SECURITY_CAPABILITIES` (appcontainer_runner.rs:188-195). That struct is attached to the

process-creation attribute list under `PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES`, the slot `CreateProcessW` reads to launch the child as an AppContainer with that package SID and enabled capabilities.

LPAC, Win32k lockdown, and the UI Job Object

Three more restrictions layer onto the same attribute list before the process runs.

LPAC (Less-Privileged AppContainer). With `least_privilege_mode` set, MXC opts out via `PROC_THREAD_ATTRIBUTE_ALL_APPLICATION_PACKAGES_POLICY` (appcontainer_runner.rs:570-587), dropping the child out of the implicit `ALL APPLICATION PACKAGES` grant and into the narrower `ALL RESTRICTED APPLICATION PACKAGES` group.

Win32k system-call disable. When the UI policy disables GUI access, the child gets the `PROCESS_CREATION_MITIGATION_POLICY_WIN32K_SYSTEM_CALL_DISABLE_ALWAYS_ON` mitigation, written through `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY` via `UpdateProcThreadAttribute` (appcontainer_runner.rs:589-610). The kernel enforces it before the child runs any user-mode code, severing the `win32k.sys` attack surface with no race window.

UI restrictions via a Job Object. Clipboard, desktop, display, handle, and atom access are restricted by encoding the policy into a `JOB_OBJECT_UILIMIT_*` bitmask:

```
pub fn to_job_object_uilimit_mask(r: &EffectiveUiRestrictions) -> u32 {
    let mut mask: u32 = 0;
    if r.block_external_ui_objects        { mask |= JOB_OBJECT_UILIMIT_HANDLES.0; }
    if r.block_clipboard_read             { mask |= JOB_OBJECT_UILIMIT_READCLIPBOARD.0
    if r.block_clipboard_write            { mask |= JOB_OBJECT_UILIMIT_WRITECLIPBOARD.(
    if r.block_system_parameter_changes   { mask |= JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS
    if r.block_display_settings_changes   { mask |= JOB_OBJECT_UILIMIT_DISPLAYSETTINGS
    if r.block_global_ui_namespace        { mask |= JOB_OBJECT_UILIMIT_GLOBALATOMS.0; }
    if r.block_desktop_switching          { mask |= JOB_OBJECT_UILIMIT_DESKTOP.0; }
    if r.block_logoff_or_shutdown         { mask |= JOB_OBJECT_UILIMIT_EXITWINDOWS.0; }
    if r.block_input_method_changes       { mask |= JOB_OBJECT_UILIMIT_IME; }
    if r.block_input_injection            { mask |= JOB_OBJECT_UILIMIT_INJECTION; }
    mask
}
```

(backends/appcontainer/common/src/job_object.rs:155-188)

The mask is intersected with `supported_ui_limit_mask()` (which build-gates the newer IME and INJECTION bits so the kernel never rejects the call on older builds), wrapped in a `JOBOBJECT_BASIC_UI_RESTRICTIONS`, and applied via `SetInformationJobObject`. A

dedicated test harness, `wxc-ui-probe`, runs *inside* a container and verifies these operations actually fail when the policy says they should.

Process creation

The child is created suspended (`EXTENDED_STARTUPINFO_PRESENT | CREATE_SUSPENDED | CREATE_UNICODE_ENVIRONMENT`) with a clean environment, resumed only once the Job Object is attached (`appcontainer_runner.rs:740-829`). That clean environment is built fresh via `CreateEnvironmentBlock(..., bInherit = FALSE)` (`appcontainer_runner.rs:76-93`), so the parent's variables never leak in. After `CreateProcessW` returns, the UI Job Object is created and assigned, and only then is `ResumeThread` called: restrictions are in force before the child's first instruction.

Network scoping

Host allow/block lists are enforced through the Windows Firewall COM API, and the key detail is *scoping*. Each rule's `SetLocalAppPackageId` binds it to the AppContainer package SID:

```
rule3
    .SetLocalAppPackageId(&BSTR::from(principal_id))
    .map_err(|e| WxcError::Firewall(format!("put_LocalAppPackageId failed: {}", e)))?;

rule.SetDirection(NET_FW_RULE_DIR_OUT) /* ... */;
rule.SetAction(action) /* ... */;
```

(`backends/appcontainer/common/src/network_manager.rs:337-404`)

`principal_id` is the package SID string, so the rule applies only to traffic from that one sandboxed package, not the whole machine.

Tier 2: the BFS broker

Tier 2 uses the entire classic-AppContainer stack from the sections above; the one thing it swaps out is filesystem enforcement. Where Tier 3 stamps ACEs onto host objects, Tier 2 registers each allowed path with the **Brokered File System (BFS)**, a Windows broker that mediates the AppContainer's file access at runtime, so the policy lives with the broker rather than on the host's security descriptors.

MXC drives BFS by shelling out to `bfscfg.exe`. Each read-write path becomes an `--addpolicy --policybroker` registration keyed to the container's `--appid`; read-only paths use a sibling function that swaps in `--policybrokerreadonly`:

```
fn add_bfs_path(&self, path: &str, inherit: bool, logger: &mut Logger) -> Result<(), W
    let mut args = vec![
        "--addpolicy",
        "--policybroker",
        "--filename",
        path,
        "--appid",
        &self.app_container_name,
    ];
    if inherit {
        args.push("--containerinherit");
    }
    let description = format!(
        "Failed to add BFS path {} for AppContainer {}",
        path, self.app_container_name
    );
    self.execute_bfscfg_operation(&args, &description, logger)
}
```

(filesystem_bfs.rs:115-132)

Every path but a bare drive root gets `--containerinherit`, so a grant carries down its subtree. Teardown is a single `--clearpolicy` for the `--appid`. Because none of this touches host security descriptors, there is nothing to undo: no per-ACE state file to replay, no orphaned grant to reap at the next boot. The `DaclManager` bookkeeping the next section describes is exactly the cost Tier 2 avoids, which is why it sits above Tier 3 in the fallback order.

The broker invocation is itself locked down: `bfscfg.exe` is resolved to its absolute path under `System32` through the Windows-directory API rather than the `SystemRoot` environment variable, then handed to `CreateProcessW` as `lpApplicationName` so the executable search order never applies. An attacker who can rewrite the process environment has no way to redirect the binary or force a Tier 2 -> Tier 3 downgrade (`fallback_detector.rs:486-507`).

All of this is behind the `tier2_bfs` Cargo feature. In a stock build it is compiled out: `find_bfscfg_exe` returns `None` before it ever hits the disk, and the detector skips from Tier 1 straight to Tier 3. So on a default binary BFS is not a fallback, it is a tier that was never built in.

Tier 3: host DACL mutation

When neither the OS API nor BFS is available, Tier 3 enforces filesystem policy bluntly: it stamps an ACE directly onto each host NTFS object, targeted at the AppContainer SID. Each

grant reads the existing DACL, merges in the new (optionally inheritable) ACE, and writes it back:

```
let ea = EXPLICIT_ACCESS_W {
    grfAccessPermissions: access_mask,   // RW_MASK or RO_MASK
    grfAccessMode: mode,                 // GRANT_ACCESS or DENY_ACCESS
    grfInheritance: ACE_FLAGS(inheritance),
    Trustee: trustee_for(&sid),          // the AppContainer SID
};
/* ... read the current DACL with GetNamedSecurityInfoW ... */
let rc = unsafe {
    SetEntriesInAclW(Some(&[ea]), Some(existing_dacl as *const ACL), &mut new_dacl)
};
/* ... */
let rc = unsafe {
    SetNamedSecurityInfoW(
        object_name, SE_FILE_OBJECT, DACL_SECURITY_INFORMATION,
        None, None, Some(new_dacl as *const ACL), None,
    )
};
```

(core/wxc_common/src/filesystem_dacl.rs:1062-1154)

The access mask is `RW_MASK` (read/write/execute/delete) for read-write paths or `RO_MASK` (read/execute) for read-only. Because this modifies *host* security descriptors, MXC must undo them: the `DaclManager` replays the inverse on drop, persists a state file per applied ACE, and reaps orphans on the next startup. The dispatcher wires this cleanup through panic unwinding and a Ctrl-C handler so a killed process doesn't leave host ACEs behind, a lot of machinery just to stop a fallback mode from leaking state onto the host.

Tier 1: the BaseContainer FlatBuffer

Tier 1 delegates all of the above to the OS in a single call. Instead of assembling attribute lists, minting capability SIDs, and stamping ACEs by hand, MXC serializes a FlatBuffer `SandboxSpec` describing the whole policy:

```
table SandboxSpec {
    version:string (required);
    app_container:bool = false;
    integrity_level:uint32 = 0 (deprecated);
    disallow_win32k_system_calls:bool = false;
    ui_restrictions:uint64 = 0;          // JOB_OBJECT_UILIMIT_* bitmask
    least_privilege:bool = false;
    capabilities:string;                 // comma-delimited names
    fs_read_write:[string];
    fs_read_only:[string];
```

```
    network_policy: NetworkPolicy;
    integrity:IntegrityLevel = system_default;
}
// ... proxy_info / NetworkPolicy tables omitted ...
root_type SandboxSpec;
file_identifier "SBOX";
```

(external/windows-sdk/BaseContainerSpecification.fbs:37-82)

Note the overlap with the manual path: `ui_restrictions` is the *same* `JOB_OBJECT_UILIMIT_*` bitmask, and `disallow_win32k_system_calls` is the same Win32k lockdown. Tier 1 hands them to the OS instead of applying them itself.

The runner loads the OS API by hand, pinned to System32 to avoid DLL planting just as the Tier 2 broker invocation was, and calls it with the spec blob:

```
let hmodule = LoadLibraryExW(
    PCWSTR(dll_name.as_ptr()),
    None,
    LOAD_LIBRARY_SEARCH_SYSTEM32,
)?;

let proc = GetProcAddress(
    hmodule,
    windows::core::PCSTR(c"Experimental_CreateProcessInSandbox".as_ptr().cast()),
)?;
```

(backends/appcontainer/common/src/base_container_runner.rs:377-409)

Its function-pointer type (base_container_runner.rs:76-91) is `CreateProcessW` with two additions: an `identity` (the AppContainer profile name) and the spec blob plus its size. The OS parses the `"SBOX"` FlatBuffer and constructs the AppContainer/LPAC token, the Win32k mitigation, the UI limits, and the filesystem ACLs from it.

The `Experimental_` prefix indicates that this interface is still in progress, but it is no longer undocumented. As of May 26th, Microsoft published `CreateProcessInSandbox` on MS Learn, a strong signal that this BaseContainer path, not the manual attribute-list assembly above, is the one headed for general availability.

Isolation Session: a per-run agent user

Isolation Session is the newest Windows backend (it needs a recent Insider build; Microsoft has said process and session isolation will be available to Windows Insiders shortly after Build) and the only one implementing the state-aware lifecycle. Rather than shape a token, it

asks the OS to provision a *separate agent user account*, runs the workload as that user, and brokers folder access back to the caller.

The whole API hangs off one WinRT object, activated through a cached factory:

```
impl IsoSessionOps {
    pub fn new() -> windows_core::Result<Self> {
        Self::IActivationFactory(|f| f.ActivateInstance::<Self>())
    }
    /* ... */
}
```

(backends/isolation_session/bindings/src/bindings.rs:2993-2996)

```
impl windows_core::RuntimeName for IsoSessionOps {
    const NAME: &'static str = "Windows.AI.IsolationSession.IsoSessionOps";
}
```

(bindings.rs:3288-3290)

There's no separate feature-flag check; the gate *is* the activation. If the OS feature is off or `IsoSessionApp.dll` isn't registered, activation returns `CLASS_E_CLASSNOTAVAILABLE`, which the manager maps to a "service unavailable" error.

The manager (manager.rs:82-96) wraps the lifecycle as a sequence of Rust methods, each one (with two exceptions) mapping 1:1 to a WinRT op on `IsoSessionOps`. A full run walks:

- `RegisterApp`: register the client (idempotent)
- `AddUserAsync`: provision a fresh agent user (the OS assigns the account name, returned via `AgentUserName()` and used for logging only; subsequent ops address the user by the `provisionId`)
- `ShareFolderBatchAsync`: share the caller's requested folders into the session. The agent runs under a different SID and inherits none of the caller's directories, so each one is granted explicitly here, after a protected-paths filter drops the dangerous top-level paths (detailed below)
- `StartSessionAsync`: start the isolation session
- `RunProcessWithOptionsAsync`: run the process. This returns a *separate* `IsoSessionProcess` object; MXC reads its stdin/stdout/stderr handles, spins up relay threads to bridge them to `wxc-exec`'s own stdio across the session boundary, and blocks on the object's `WaitForExit` (a kernel wait, not a COM call)
- `StopSessionAsync`, then `RemoveUserAsync`: stop the session and deprovision the agent user

- The OS also exposes `UnregisterAppAsync`, but MXC's `unregister_client` wrapper deliberately skips it: the `"regid"` registration is shared across all of the caller's concurrent sandboxes, so unregistering would tear down the others

The term "isolation" is used here in a different sense than the rest of the backends. `build_iso_process_options` sets only stdio redirection, working directory, timeout, and environment (process_options.rs:95-136), and the command then runs as the OS-provisioned agent user with whatever token the broker hands it. The boundary is the Windows account itself. Because the agent gets a different SID than the caller, Windows' security model denies it the caller's per-user state by default: the user profile, the `HKCU` hive, user-scoped DPAPI secrets, the logon token.

Beyond that the picture is undocumented. The repo never sets the agent's token, so its exact privilege level, whether it's an ordinary local account that can reach anything `Users` / `Authenticated Users` can, or a more restricted principal, is entirely the OS-side broker's call. The broker itself lives in an unreleased OS component (`IsoSessionApp.dll`), so the specifics will have to wait for the Windows Insider preview that ships it. What MXC actually provides is narrower: a separate principal with no shared per-user state, and that property is why the caller's own folders have to be shared back in explicitly.

For Microsoft Entra cloud agents, a v2 path (`AddUserAsync2` / `StartSessionAsync2`) forwards a short-lived WAM bearer token to the OS service. MXC stores nothing, and the credential type has a hand-written `Debug` impl that redacts the token:

```
impl std::fmt::Debug for IsolationSessionUser {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.debug_struct("IsolationSessionUser")
            .field("upn", &self.upn)
            .field("wam_token", &"<redacted>")
            .finish()
    }
}
```

(core/wxc_common/src/models.rs:227-244)

Sharing folders into the session

Because the agent runs as a different user, the caller's directories must be explicitly shared in. That happens through `ShareFolderBatchAsync`, wrapped by a filter worth quoting in full:

```
//! Emergency mitigation (MXC issue #330) — silently drops a fixed set of
//! system-folder paths from filesystem-policy requests before they reach
//! `ShareFolderBatchAsync`. The OS API applies grants with subtree
//! inheritance, so granting these top-level folders propagates an agent
```

```
//! SID's ACE through their entire subtrees — catastrophic for drive roots,
//! the Windows directory, the Users root, ProgramFiles, and ProgramData.
//!
//! The proper fix belongs in the OS API. When that lands, delete this
//! file and the call site in `manager.rs::share_folders`.
//!
//! Known not covered (caller would have to actively pursue these spellings
//! to evade the text match): 8.3 short names (e.g. `PROGRA~1`), symlinks /
//! junctions (no `canonicalize` disk access), UNC paths,
//! `CommonProgramFiles` / `CommonProgramFiles(x86)`. The `\\?\` long-path
//! prefix on a disk path (`\\?\C:\foo`) IS handled — it normalizes to the
//! same canonical form as `C:\foo` so it cannot be used as a textual bypass.
```

(backends/isolation_session/common/src/protected_paths_filter.rs:4-19)

The mechanism is a deny-set: all 26 drive roots plus canonicalized system environment paths (`SystemRoot`, the parent of `USERPROFILE`, the Program Files variants, `ProgramData`). Share requests for those paths are dropped before they reach the OS API. The comment is blunt about why: the OS grant inherits down the whole subtree, so sharing a top-level folder would hand the agent far more than intended, and this user-mode filter is a stopgap until the OS API grows a narrower grant.

Windows Sandbox: a disposable Hyper-V VM

The `windows_sandbox` backend is the heaviest Windows tier: a full, throwaway Windows VM via the built-in Windows Sandbox feature. Three components: the `wxc-exec` client, a long-lived host daemon that manages the VM, and a guest agent that runs inside it.

The VM is configured by generating a `.wsb` file:

```
let wsb_content = format!(
    r#"<Configuration>
  <MappedFolders>
    <MappedFolder>
      <HostFolder>{host_guest}</HostFolder>
      <SandboxFolder>{sandbox_guest}</SandboxFolder>
      <ReadOnly>true</ReadOnly>
    </MappedFolder>
    /* rendezvous (rw) and python (ro) folders ... */
  </MappedFolders>
  <LogonCommand>
    <Command>{sandbox_rendezvous}\bootstrap.cmd</Command>
  </LogonCommand>
  <vGPU>Disable</vGPU>
  <Networking>Enable</Networking>
</Configuration>"#,
```

```
    /* ... */
);
```

(backends/windows_sandbox/daemon/src/sandbox_vm.rs:163-201)

The guest binary and a host Python install are mapped read-only; a read-write "rendezvous" directory carries the bootstrap script and logs. Host and guest talk over **four separate TCP connections**, one for control plus one each for stdin, stdout, and stderr:

```
/// Four TCP connections to the guest agent.
///
/// TODO: These TCP connections are unencrypted. Verify if this is a concern.
pub struct GuestConnection {
    pub control: TcpStream,
    pub stdin_stream: TcpStream,
    pub stdout_stream: TcpStream,
    pub stderr_stream: TcpStream,
}
```

(backends/windows_sandbox/daemon/src/tcp_bridge.rs:23-31)

The `.wsb` enables networking so the channels can form, then the *guest* immediately locks the network down to the host alone:

```
/// 1. Delete all existing rules.
/// 2. Allow inbound on our listen port from the host IP.
/// 3. Allow outbound to the host IP.
/// 4. Set default policy to block-all for both directions.
pub async fn lockdown(host_ip: IpAddr, listen_port: u16) -> Result<()> {
    /* netsh advfirewall delete rule name=all */
    /* netsh advfirewall add rule ... remoteip=<host> ... */
    /* netsh advfirewall set allprofiles firewallpolicy blockinbound,blockoutbound */
}
```

(backends/windows_sandbox/guest/src/firewall.rs:11-70)

The isolation here is the VM boundary itself: a separate Windows OS instance with its own filesystem, registry, and network stack, destroyed on teardown. The source flags two preview-grade rough edges: teardown uses `taskkill /F /IM`, which is system-wide, and the VM is reused across executions, so one script's side effects can persist into the next.

WSLC: Linux containers from the Windows front-end

The WSLC SDK this backend rides on is itself one of the Build 2026 announcements: Microsoft pitched WSL containers as "a built-in way to create, run and interact with Linux

containers on Windows," due in public preview "in the coming months as a regular update to WSL." So this is a sneak-peek at an API that is not in users' hands yet.

The `wslc` backend runs OCI/Linux containers inside a WSL2 micro-VM, driven by Microsoft's first-party WSLC SDK (a three-level `Session -> Container -> Process` handle model). Each host path is translated to a WSL mount point (`C:\workspace` becomes `/mnt/c/workspace`) and handed to the SDK as a `WslcContainerVolume`:

```
let volumes: Vec<WslcContainerVolume> = wide_paths
    .iter()
    .zip(mounts.iter())
    .map(|((win, ctr), m)| WslcContainerVolume {
        windows_path: win.as_ptr(),
        container_path: ctr.as_ptr() as PCSTR,
        read_only: if m.read_only { 1 } else { 0 },
    })
    .collect();

let hr = (sdk.WslcSetContainerSettingsVolumes)(
    &mut container_settings,
    volumes.as_ptr(),
    volumes.len() as u32,
);
```

(backends/wslc/common/src/wsl_container_runner.rs:997-1011)

The per-volume `read_only` flag carries the read-only/read-write policy straight into the container settings. Networking maps to one of two SDK modes:

```
pub fn map_network_policy(is_block: bool, has_host_rules: bool) -> WslcContainerNetworl
    if is_block && !has_host_rules {
        WslcContainerNetworkingMode::None       // no interface, fully isolated
    } else {
        WslcContainerNetworkingMode::Bridged    // NAT through the WSL2 VM
    }
}
```

(policy_mapping.rs:122-128)

Per-host filtering, when requested, builds an iptables rule chain and exec's it inside the started (privileged) container via `WslcCreateContainerProcess`, so the container image must ship iptables and run with `NET_ADMIN`. The rule builder validates host strings against shell metacharacters before interpolation, and child stdout/stderr arrive through an `io_callback` tagged with a `WslcProcessIOHandle` (Stdin/Stdout/Stderr) enum rather than pulled from a getter.

NanVix and Hyperlight: the micro-VM tiers

Two backends run code in lightweight VMs rather than OS containers.

NanVix boots a minimal guest OS over the Windows Hypervisor Platform (WHP, with a warm-start memory snapshot) or KVM on Linux (cold boot each time). It never mounts host directories live. Instead it *copies* them into a per-run staging tree, hands that tree to the guest as a single `-mount` argument on the `nanvixd` command line, and copies the writable files back afterward:

```
cmd.current_dir(&paths.snapshot_home)
    .arg("-snapshot")
    .arg(&snapshot_rel)
    .arg("-bin-dir")
    .arg(paths.exe_dir.join(BIN_DIR))
    .arg("-ramfs")
    .arg(&paths.ramfs)
    .arg("-mount")
    .arg(staging_dir)
    .arg("--")
    .arg(&paths.initrd);
```

(backends/nanvix/runner/src/lib.rs:531-541)

The staging tree is presented to the guest under `/mnt/rw/...`, and the user's script is rewritten accordingly. Copy-back runs only on a clean exit:

```
/// Copyback runs on any normal process exit (including non-zero exit codes).
/// It is skipped for preflight, spawn, runtime, and timeout errors, and for
/// OS crashes (negative exit codes from NTSTATUS values).
fn should_copy_back(response: &ScriptResponse) -> bool {
    response.error_message.is_empty() && response.exit_code >= 0
}
```

(backends/nanvix/runner/src/lib.rs:750-756)

Hyperlight is the strangest of the lot: it runs CPython *in-process* inside a Unikraft unikernel on a Hyperlight micro-VM, no subprocess and no pipes. Code is delivered by a direct library call, and each `run_code` rewinds the guest to a post-warmup snapshot so consecutive runs are hermetic. Host directories are exposed as `Preopen` mounts at `/host/<basename>`, with the policy's read-only flag mapped straight onto `Preopen::read_only()`:

```
let guest_path = format!("/host/{basename}");
/* ... reject guest-path collisions ... */
let mut pre = Preopen::new(&host_path, &guest_path).map_err(|e| {
    PyhlError::Preflight(format!(
```

```
        "build Preopen for {host:?} -> {guest_path:?}: {e:#}"
    ))
})?;
if read_only {
    pre = pre.read_only();
}
preopens.push(pre);
```

(backends/hyperlight/common/src/lib.rs:404-419)

The returned timing splits `restore_ms` (the snapshot rewind) from `call_ms` (the run). The micro-VM mechanics live in the external `pyhl` / `hyperlight-unikraft` crate, not the MXC repo.

Bubblewrap: unprivileged Linux namespaces

On Linux the default backend is `bubblewrap`, which builds an argument vector for the `bwrap` tool: unprivileged Linux namespace isolation, no root required. The whole sandbox is described declaratively:

```
let mut args = vec![
    "--unshare-user",
    "--unshare-pid",
    "--unshare-ipc",
    "--unshare-uts",
] /* ... */;
/* ... */
args.extend(["--ro-bind".into(), "/".into(), "/".into()]);
args.extend(["--dev".into(), "/dev".into()]);
args.extend(["--proc".into(), "/proc".into()]);
args.extend(["--tmpfs".into(), "/tmp".into()]);

for path in &request.policy.readwrite_paths {
    args.extend(["--bind".into(), path.clone(), path.clone()]);
}
for path in &request.policy.readonly_paths {
    args.extend(["--ro-bind".into(), path.clone(), path.clone()]);
}
for path in &request.policy.denied_paths {
    args.extend(["--tmpfs".into(), path.clone()]);   // mask with empty tmpfs
}
```

(backends/bubblewrap/common/src/bwrap_command.rs:38-144)

The ordering is deliberate: the read-only root and standard virtual filesystems go down first, policy mounts last so they win on overlap. Read-write paths are bind mounts, read-only paths are `--ro-bind`, denied paths are masked with an empty tmpfs.

Networking has two modes. With a pure block and no host rules, `--unshare-net` gives a zero-cost, fully isolated network namespace. For per-host filtering, the namespace stays shared and a cooperative loopback HTTP proxy is used instead. A comment explains why `NO_PROXY` is deliberately *not* set:

```
// We deliberately do NOT set NO_PROXY here. Bubblewrap with a proxy
// keeps the host network namespace shared, so without a NO_PROXY entry
// a cooperating client doing `CONNECT 127.0.0.1:5432` (e.g. local
// Postgres) still goes via the proxy, where the configured
// allowed/blocked-hosts policy applies.
```

(bwrap_command.rs:123-127)

The proxy is cooperative by design: well-behaved clients honor the injected `HTTP_PROXY`/`HTTPS_PROXY`, while raw-socket clients slip past unfiltered, a limitation the source documents. iptables-based enforcement (which needs `CAP_NET_ADMIN`) and the proxy are mutually exclusive, rejected at config-parse time.

LXC: system containers

The `lxc` backend drives full LXC system containers. Despite a docstring referencing "liblxc," it shells out to the `lxc-*` CLI tools with a consistent `-P <path> -n <name>` prefix. The lifecycle is create -> start -> attach-run -> destroy:

```
pub fn create(&self, distribution: &str, release: &str) -> Result<(), String> {
    let mut cmd = self.lxc_command("lxc-create");
    cmd.args(["-t", "download", "--", "-d"])
        .arg(distribution)
        .arg("-r").arg(release)
        .arg("-a").arg(Self::current_arch());
    Self::run_status(cmd, "lxc-create")
}
```

(backends/lxc/common/src/lxc_bindings.rs:147-156)

The `download` template pulls a prebuilt rootfs for the chosen distro/release/arch (e.g. `alpine` / `3.23`). Filesystem policy becomes `lxc.mount.entry` lines: `bind,create=dir` for read-write, `bind,ro,create=dir` for read-only, and a `/dev/null` bind (files) or zero-size tmpfs (directories) to mask denied paths.

Network filtering builds a per-container iptables chain named `MXC-<name>` and scopes it to the container's host-side veth interface, discovered by parsing `lxc-info`:

```
if let Some(ref iface) = self.veth_interface {
    Self::run_iptables(&["-I", "FORWARD", "-o", iface, "-j", &self.chain_name], logger
```

```
} else {
    // Without a veth interface, we cannot safely scope rules to the container.
    // Refuse to apply host-wide rules to avoid affecting all host traffic.
}
```

(backends/lxc/common/src/network_iptables.rs:183-234)

If the veth can't be found, the FORWARD hook is skipped rather than applied host-wide: fail closed on scoping rather than risk touching all host traffic.

Seatbelt: generating a macOS sandbox profile

On macOS, the `seatbelt` backend wraps the kernel sandbox behind the App Sandbox. MXC generates a TinyScheme SBPL profile, deny-by-default, and applies it with `sandbox_init()`. The profile is assembled in a careful order:

```
out.push_str("(version 1)\n");
out.push_str("(deny default)\n");
out.push_str(BASELINE_ALLOW);       // fork/exec, signal self, sysctl, basic mach serv
out.push_str(SYSTEM_READ_ALLOW);    // read of /usr/lib, /System, /Library, dyld cache
out.push_str(TTY_ALLOW);
write_filesystem_allow(&mut out, request)?;   // readonly -> file-read*, readwrite -> 
write_network_rules(&mut out, request);
write_nested_pty_rules(&mut out, request);
write_keychain_rules(&mut out, request)?;
write_extra_seatbelt_rules(&mut out, request);
write_ui_rules(&mut out, request);
write_filesystem_deny(&mut out, request)?;    // deniedPaths LAST so they win
```

(backends/seatbelt/common/src/profile_builder.rs:38-82)

Seatbelt is last-match-wins within an operation, so emitting `deniedPaths` last guarantees they override broader allows. Filesystem policy maps cleanly: read-only paths become `(allow file-read* (subpath ...))`, read-write add `file-write*`, denied paths become `(deny file-read* file-write* (subpath ...))`. UI restrictions deny mach-lookups of WindowServer / LaunchServices, the pasteboard service for clipboard, and `IOHIDLibUserClient` for injection.

One detail diverges from naive expectation: `allowedHosts` does *not* produce per-host network rules. Seatbelt can't filter outbound by hostname, so an allowlist degrades to best-effort allow-all, and `blockedHosts` is rejected outright at validation rather than silently ignored.

The profile is applied in the child between fork and exec:

```
command.pre_exec(move || {
    let mut errorbuf: *mut libc::c_char = std::ptr::null_mut();
```

```
    let rc = sandbox_init(profile_cstr.as_ptr(), 0, &mut errorbuf);
    if rc != 0 {
        /* write error to fd 2 with libc only, no allocation */
        return Err(std::io::Error::from_raw_os_error(libc::EPERM));
    }
    Ok(())
});
```

(backends/seatbelt/common/src/seatbelt_runner.rs:383-423)

The comment notes this fork -> `sandbox_init()` -> exec pattern is the one Chromium uses. The closure is async-signal-safe (a pre-built CString, libc-only error reporting) because it runs in the fragile window after fork.

Cross-cutting design notes

Reading all ten backends together, a few patterns stand out. These are observations about the design, not judgments about readiness.

The policy model is the common thread. Every backend consumes the same cross-platform policy (filesystem read-only/read-write/denied, network posture, UI restrictions) and translates it into native enforcement. The same `JOB_OBJECT_UILIMIT_*` bitmask appears in the manual AppContainer path *and* the BaseContainer FlatBuffer; the same read-only/read-write/denied triple becomes DACL ACEs on Windows, bind mounts on bubblewrap, `lxc.mount.entry` lines on LXC, SBPL subpaths on macOS, and staged file copies on NanVix.

Default-deny is consistent. UI policy defaults to disabled, no clipboard, no injection; network defaults to block; deny rules are honored last where supported. The Seatbelt and bubblewrap profiles are deny-by-default with explicit allows.

Enforcement strength varies by tier, and the source says so. Some boundaries are kernel-enforced (AppContainer tokens, Win32k disable, Job Object UI limits, namespace unsharing, Seatbelt, the Hyper-V VM boundary). Others are cooperative (the bubblewrap and Windows proxy paths filter only clients that honor proxy env vars; raw sockets bypass them). The code is explicit about which is which, in comments next to the mechanism.

Unix backends share PTY plumbing. LXC and the Seatbelt CLI path both go through `mxc_pty::run_with_pty`, which allocates a pty pair, runs `setsid()` + `TIOCSCTTY` in a post-fork `pre_exec`, and gates stdin forwarding on a first-byte "ready" handshake to avoid racing the inner shell's terminal setup.

Closing thoughts

Every other vendor in the intro builds its sandbox at the application layer: Codex shells out to Seatbelt and Landlock, Cowork ships its own Linux VM, the hosted services run your code inside their containers. MXC is the OS vendor answering the same question from underneath. It is not one sandbox but a dispatcher that meets each platform where its isolation primitives already live: AppContainer and the new BaseContainer API on Windows, a per-run agent user through the isolation-session broker, disposable Hyper-V VMs, WSL containers, two flavors of micro-VM, namespaces on Linux, Seatbelt on macOS. One policy model feeds all of them, and most of the work is the translation down to each OS's native enforcement, plus the unglamorous job of cleanly undoing host state when a fallback tier had to touch it.

The "agentic" isolation framing is of the moment, but the design is versatile. Underneath the README's language about model output and tools is a general untrusted-code container: policy in, confined process out. A model is one source of code you don't trust, but so is a third-party plugin, a CI job running someone's pull request, or a script a user pasted in. Nothing in the dispatcher knows an agent produced the bytes, and nothing would change if one hadn't. That is the part I keep coming back to: MXC is a window into how Microsoft is thinking about running untrusted code, with agents as the reason it shipped now.

It is an early preview, and the source is honest about the rough edges: the `Experimental_` prefix, the issue-#330 folder-share filter, the unencrypted sandbox TCP channels, the cooperative-only proxies. None of it is hidden; it sits in the comments next to the code. Two things make it worth reading now anyway. The OS surface underneath is starting to come out into the open, like the newly documented `CreateProcessInSandbox`, and the freshest backends are a preview of where Windows containment is heading, isolation-session and BaseContainer most of all. If you build or attack this kind of infrastructure, that is the payoff here: a first-party, read-the-source look at how a platform vendor plans to contain code it didn't write, well before the polished version lands.