

WPAR (Container) Detection for AIX Events - Research Findings

📄 This page just contains the research right now. Implementation and testing is TBD.

Last updated: Jan 26 @ 12:00PM.

Overview

This document summarizes research into whether we can determine if a process/file/port event on AIX originated from inside a WPAR (Workload Partition - AIX's container technology) or from the Global system.

TL;DR: Yes, it's possible. AIX provides mechanisms to identify WPAR origin at the process level.

Background

When we receive an event (e.g., "user ran `ls` as root"), we need to determine:

- Did this happen in the Global AIX environment (the host)?
- Or did this happen inside a WPAR (container)?

This is important for security context and event correlation.

WPAR Identifier Types

Each WPAR has multiple identifiers:

Identifier	Description	How to Get	Notes
WPAR Name	Human-readable name assigned at creation	<code>lswpar</code>	e.g., "webapp_prod"
WPAR Key	Static numeric ID, same across restarts	<code>uname -W</code>	0 = Global environment
Configured ID	Dynamic ID, assigned fresh at each start	<code>lparstat -W</code>	0 = Global environment
UUID	Universally unique identifier	<code>lswpar -a UUID <name></code>	Persists across restarts

Kernel CID	Index in kernel WPAR table	Internal (see <code>/etc/corrals/index</code>)	Used by kernel internally

Reference: [IBM Docs: Listing WPAR Identifiers](#)

Method 1: Process-Level Identification (from Global Environment)

From the Global AIX environment, we can see which WPAR any process belongs to:

```

1 # Show all processes with WPAR column
2 ps -ef@
3
4 # Filter to specific WPAR only
5 ps -ef@ webapp_prod
6
7 # Filter to Global environment only
8 ps -ef@ Global
9

```

Output includes a WPAR column showing:

- `Global` - process running in Global environment
- `<wpar_name>` - process running in that specific WPAR

Important: Running `ps -@` from *inside* a WPAR always shows "Global" because all processes appear global from that WPAR's perspective.

Reference: [IBM: How to Differentiate between Global Environment and WPAR processes](#)

Method 2: Self-Detection (Is This Process in a WPAR?)

A process can determine if it's running inside a WPAR:

```

1 #!/bin/bash
2 wpar_id=$(uname -W)
3 if [ "$wpar_id" -eq 0 ]; then
4     echo "Running in Global AIX"
5 else
6     echo "Running in WPAR #$wpar_id"
7 fi

```

Reference: [IBM: Am I a WPAR or a regular AIX VM?](#)

Method 3: Kernel-Level (Programmatic)

What We Know

The kernel tracks WPAR membership via a **Corral ID (CID)**:

- Each process has an associated CID in the kernel process structure
- CID 0 = Global environment
- CID > 0 = specific WPAR
- The mapping of CID to WPAR name is stored in `/etc/corrals/index`

File format of `/etc/corrals/index` :

```
1 WparID:Type:Kernel_CID_value
```

Example:

```
1 1:S:wpar1:1
```

Where **S** = System WPAR, **A** = Application WPAR, **L** = Versioned WPAR

Relevant Header Files

Header	Purpose
<code>/usr/include/procinfo.h</code>	Contains <code>procentry64</code> structure used by <code>getprocs64()</code>
<code>/usr/include/sys/corralid.h</code>	Contains WPAR-related macros (per IBM APAR IZ95987)

What Needs Verification on AIX

The exact field name for WPAR/CID in the `procentry64` structure is not publicly documented. We need to check on our AIX 7.2 system:

```
1 # Check procentry64 for WPAR fields
2 grep -i "cid\|wpar\|corral" /usr/include/procinfo.h
3
4 # Check the corral header
5 cat /usr/include/sys/corralid.h
6
7 # View WPAR index file
8 cat /etc/corrals/index
9
```

Reference: [IBM APAR IZ95987 - sys/corralid.h](#)

Implications for Metadata Collector

ProcessCollector

We can add WPAR identification by:

1. **Parsing `ps -ef@` output** - Most straightforward, works today
2. **Using the CID field in `procentry64`** - Requires finding the exact field name in the header
3. **Mapping CID to WPAR name** - Read `/etc/corrals/index` for the mapping

PortCollector

The process associated with a network port inherits its WPAR context. If we identify the owning process's WPAR, we know the port's WPAR.

FileCollector

Files in WPARs typically reside under `/wpars/<wpar_name>/` mount points, which can be detected from the file path.

Visibility Rules

Observer Location	Can See
Global Environment	All processes (Global + all WPARs)
Inside WPAR	Only processes within that WPAR

This means our metadata collector running in Global can see everything, but would need elevated visibility if running inside a WPAR.

Next Steps

1. **Verify on AIX 7.2 VM** - Check `/usr/include/procinfo.h` for the exact CID field name
2. **Prototype** - Add WPAR field to ProcessCollector using `ps -@` parsing as a first pass
3. **Optimize** - If we find the CID field, use `getprocs64()` directly for better performance

References

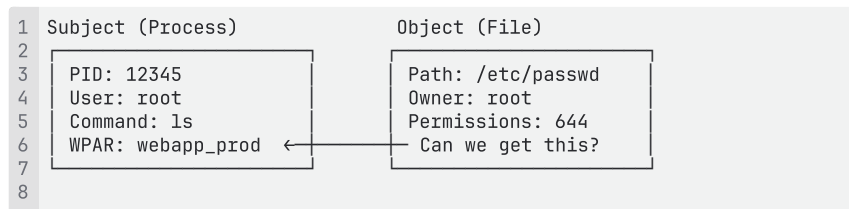
Resource	URL
IBM: Differentiate Global vs WPAR processes	https://www.ibm.com/support/pages/how-differentiate-between-global-environment-and-wpar-processes-aix-61-and-higher
IBM: Am I a WPAR?	Am I a WPAR? (workload partition) or a regular AIX VM (virtual machine)?
IBM: Listing WPAR Identifiers	Listing WPAR identifiers
IBM: AIX WPARs How-To	AIX WPARs - How to
IBM APAR IZ95987 (sys/corralid.h)	IZ95987: MOVE GET_NOVIP_WPAR MACRO TO CORRAL HEADER FILE APPLIES TO AIX 6100-07
WPAR FAQ	(FAQ) Frequently Asked Questions on WPAR (Workload Partitions)
Creating a System WPAR	Creating a system WPAR
[Full Playlist] Introduction to AIX 6 WPARs	AIX Workload Partition (WPAR) - part1 - Introduction (from 2008)

Summary for Subject (Process) Context

Method	How It Works	Use Case
<code>uname -W</code>	Returns WPAR key (0 = Global, >0 = WPAR)	Self-detection: "Am I in a WPAR?"
<code>ps -ef@</code>	Shows WPAR column for all processes	From Global: "Which WPAR is PID X in?"
Kernel CID	Process structure contains Corral ID	Programmatic: <code>getprocs64()</code> may expose this field
<code>/etc/corrals/index</code>	Maps CID → WPAR name	Translate numeric CID to WPAR name

For Our Use Case (Subject/Object Model)

When we receive an event like "process X accessed file Y":



From the Global environment, given a PID, we can determine its WPAR membership via:

1. **Quick method:** Parse output of `ps -o wpar= -p <PID>`
2. **Programmatic:** Use `getprocs64()` and read the CID field (need to verify exact field name on AIX)
3. **Self-check:** If the subject process calls `uname -W`, it gets its own WPAR ID

Key Point

The subject process's WPAR context is tracked by the kernel and is queryable. This means for security event correlation, we **can** enrich events with container context.

Iteration 1: Run `ps -o wpar=` command to get data

<https://dsgithub.trendmicro.com/kien-do/aix-metadata-collector-poc/pull/1>

Iteration 1 - Claude Code Notes

Summary: WPAR Container Detection Implementation

What I've Done

1. Updated ProcessCollector with WPAR Detection

I've added two new methods to `process_collector.h` and `process_collector.cpp`:

- `collectWparInfo(pid_t pid, MetadataResult& result)` - Detects if a process is from a WPAR container
- `executeCommand(const std::string& cmd)` - Helper to run shell commands and capture output

2. New Metadata Fields

The ProcessCollector now outputs these additional fields when you query a process:

Field	Description	Example Values
<code>wpar_name</code>	Name of the WPAR or "Global"	"mywpar", "Global"
<code>is_container</code>	Whether process is in a container	"true", "false"
<code>wpar_id</code>	WPAR numeric ID (if in container)	"1", "2"
<code>wpar_type</code>	Type of WPAR	"system", "application", "versioned"
<code>wpar_kernel_c id</code>	Kernel Corral ID	"1", "2"
<code>wpar_config</code>	Path to WPAR config file	"/etc/corrals /mywpar.cf"

3. Example Output

Process in Global environment:

```
1 {
2   "success": true,
3   "type": "process",
4   "identifier": "12345",
5   "attributes": {
6     "pid": "12345",
7     "comm": "bash",
8     "wpar_name": "Global",
9     "is_container": "false",
10    ...
11  }
12 }
13
```

Process inside a WPAR:

```
1 {
```

```
2  "success": true,
3  "type": "process",
4  "identifier": "67890",
5  "attributes": {
6    "pid": "67890",
7    "comm": "httpd",
8    "wpar_name": "webapp_prod",
9    "is_container": "true",
10   "wpar_id": "1",
11   "wpar_type": "system",
12   "wpar_kernel_cid": "1",
13   "wpar_config": "/etc/corrals/webapp_prod.cf",
14   ...
15  }
16 }
17
```

How to Test on AIX

Step 1: Create a Test WPAR

```
1  # Create a simple system WPAR (requires root)
2  mkwpar -n testwpar -N interface=en0 address=<IP>
   netmask=255.255.255.0
3
4  # Or create a rootvg WPAR on a dedicated disk
5  mkwpar -D devname=hdisk1 rootvg=yes -l -n testwpar
6
7  # Start the WPAR
8  startwpar testwpar
9
10 # Verify it's running
11 lswpar
12
```

Step 2: Run a Process Inside the WPAR

```
1  # Login to the WPAR
2  clogin testwpar
3
4  # Inside the WPAR, run some processes
5  sleep 3600 &
6  ps -ef
7
8  # Note the PIDs of processes running inside
9  # Exit back to Global
10 exit
11
```

Step 3: Test the Metadata Collector from Global

```
1  # From Global environment, query a WPAR process
2  ./bin/aix-metadata-collector --process <PID_from_wpar>
3
4  # Should show:
5  # "wpar_name": "testwpar"
6  # "is_container": "true"
7
8  # Query a Global process for comparison
9  ./bin/aix-metadata-collector --process 1
10
11 # Should show:
12 # "wpar_name": "Global"
13 # "is_container": "false"
14
```

Step 4: Verify with ps Command

```
1  # Compare with ps output
2  ps -ef@
3
4  # The WPAR column should match what the collector reports
5
```

Alternative: Use wparexec for Quick Testing

If you don't want to create a persistent WPAR:

```
1 # Run a command in a temporary application WPAR
2 wparexec -n tempwpar -- /usr/bin/sleep 300 &
3
4 # Quickly find the PID
5 ps -ef@ | grep sleep
6
7 # Test the collector
8 ./bin/aix-metadata-collector --process <PID>
9
```

The application WPAR will auto-destroy when the sleep command finishes.

Key Files Changed

File	Changes
include/process_collector.h	Added <code>collectWparInfo()</code> and <code>executeCommand()</code> declarations
src/process_collector.cpp	Added WPAR detection implementation

Next Steps for You

1. **Transfer to AIX:** Copy the updated files to your AIX VM
2. **Build:** Run `make` on AIX
3. **Create Test WPAR:** Use the commands above
4. **Test:** Run the collector against both Global and WPAR processes
5. **Verify procentry64:** Check `/usr/include/procinfo.h` for any `pi_cid` field for future optimization

Optional Optimization

The current implementation uses `ps -o wpar=` which spawns a subprocess. If you find a CID field in `procentry64`, you can avoid this by reading it directly:

```
1 # On AIX, check for CID field
2 grep -i "cid\|wpar\|corral" /usr/include/procinfo.h
3
```

If found, I can update the code to use the native API instead of spawning `ps`.

Sources

- [IBM: How to Differentiate Global vs WPAR processes](#)
- [IBM: AIX WPARs How-To](#)
- [IBM: Creating a System WPAR](#)
- [WPAR FAQ](#)

▾ What about method 2?

This implementation uses **Method 1: Parse** `ps -o wpar= output`.

Looking at line 483 of process_collector.cpp:

```

1 cmd << "ps -o wpar= -p " << pid << " 2>/dev/null";
2

```

This spawns the `ps` command and parses its output to get the WPAR name.

Why Method 1?

Method	Approach	Status
Method 1 ✓	Parse <code>ps -o wpar= output</code>	Implemented - Reliable, works today
Method 2	Use CID field in <code>procentry64</code>	Not implemented - Need to verify field exists in <code>/usr/include/procinfo.h</code> on AIX
Method 3	Read <code>/etc/corrals/index</code> and map CID	Partially used (for extra metadata like <code>wpar_type</code> , <code>wpar_id</code>)

Trade-offs

Method 1 (current):

- ✓ Reliable - uses same mechanism as `ps -@`
- ✓ Works without knowing internal kernel structures
- ✗ Spawns a subprocess for each query (slower)

Method 2 (future optimization):

- ✓ No subprocess - reads directly from `getprocs64()`
- ✓ Faster
- ✗ Need to verify the field name exists (e.g., `pi_cid` or similar)

To Enable Method 2

On your AIX machine, run:

```
1 grep -iE "cid|wpar|corral" /usr/include/procinfo.h
2
```

If you find a field like `pi_cid` in the `procentry64` structure, let me know and I can update the code to use the native API instead of spawning `ps`.

Iteration 2: Use procInfo.h

<https://dsgithub.trendmicro.com/kien-do/aix-metadata-collector-poc/pull/2>

Iteration 2 - Claude Code Notes

Removed

- `executeCommand()` method - no longer spawn shell processes
- All `popen()` / `pclose()` calls
- All `ps`, `grep`, `uname` command parsing

New Implementation

Now uses **Method 2: Direct** `pi_cid` from `procentry64`

```
1 struct procentry64 procInfo;
2 pid_t inputPid = pid;
3 getprocs64(&procInfo, sizeof(procInfo), nullptr, 0, &inputPid, 1);
4
5 cid_t wparCid = procInfo.pi_cid; // Direct kernel value!
6
7 if (wparCid == 0) {
8     // Global environment
9 } else {
10     // In a WPAR with CID = wparCid
11 }
12
```

WPAR Name Resolution

If `pi_cid > 0`, we read `/etc/corrals/index` (file I/O, not command execution) to resolve the WPAR name. This is a simple file read, not spawning a subprocess.

New Output Fields

Field	Description	Example
<code>wpar_cid</code>	Kernel Corral ID (0 = Global)	<code>0</code> , <code>1</code> , <code>2</code>
<code>is_container</code>	Whether process is in a container	<code>"true"</code> , <code>"false"</code>

<code>wpar_name</code>	WPAR name (only if in container)	<code>"mywpar"</code>
<code>wpar_id</code>	WPAR numeric ID (only if in container)	<code>"1"</code>
<code>wpar_type</code>	WPAR type (only if in container)	<code>"system"</code> , <code>"application"</code> , <code>"versioned"</code>

Pros and Cons

Aspect	Method 1 (Old: <code>ps</code> command)	Method 2 (New: <code>pi_cid</code>)
Speed	Slow - spawns subprocess	Fast - direct struct read
Accuracy	Subject to output parsing bugs	Direct kernel value
Dependencies	Requires <code>ps</code> command	None (kernel API)
Overhead	Fork + exec + pipe	Single <code>getprocs64()</code> call (already made)
WPAR Name	Returned directly	Requires <code>/etc/corrals/index</code> lookup
Portability	Works if <code>ps</code> supports <code>-o wpar</code>	Requires <code>pi_cid</code> in struct

Expected Output (After Fix)

For a process in **Global**:

```
1 {
2   "wpar_cid": "0",
3   "is_container": "false"
4 }
5
```

For a process in a **WPAR**:

```
1 {
2   "wpar_cid": "1",
3   "is_container": "true",
4   "wpar_name": "mywpar",
5 }
```

```

5     "wpar_id": "1",
6     "wpar_type": "system"
7 }
8

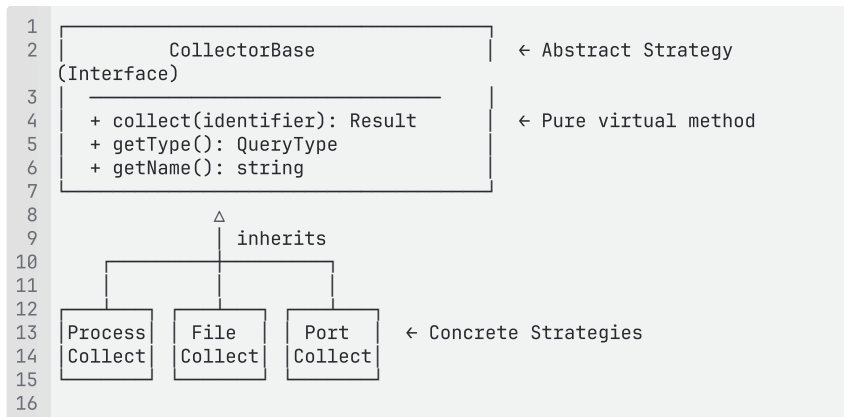
```

Design Pattern

- Developed a system utility using AIX kernel APIs (getprocs64, proccentry64) to collect process, file, and network metadata with JSON output for security event correlation.
- Implemented WPAR container detection by reading kernel Corral ID from process structures, enabling identification of containerized vs global processes.
- Designed extensible collector architecture using the Strategy pattern, allowing new metadata types to be added without modifying existing code.

The **Strategy pattern** is used here to allow different metadata collection algorithms to be selected at runtime based on user input.

Structure



How It Works

1. **Abstract Interface** (collector_base.h): Defines `collect()` as a pure virtual method that all collectors must implement.
2. **Concrete Strategies**: Each collector encapsulates its own algorithm:
 - `ProcessCollector` → uses `getprocs64()`, reads `/proc/[pid]/`
 - `FileCollector` → uses `stat64()`, `lstat64()`
 - `PortCollector` → parses `netstat -Aan` output
3. **Context Selection** (main.cpp): The switch statement picks the strategy:

```

1 switch (args.mode) {
2     case Mode::Process: {
3         ProcessCollector collector;
4         result = collector.collect(args.identifier);
5         break;
6     }
7     case Mode::File: {
8         FileCollector collector;
9         result = collector.collect(args.identifier);
10        break;

```

```
11     }  
12     // ...  
13 }  
14
```

Benefits

- **Open/Closed Principle:** Add new collectors (e.g., `UserCollector`, `ServiceCollector`) without modifying existing code
- **Single Responsibility:** Each collector handles only one type of metadata
- **Testability:** Collectors can be unit tested independently
- **Runtime Selection:** User chooses the algorithm via `--process`, `--file`, or `--port` flags