Sharpening Our Arrows: Training with Apollo

Dwight Hohnstein

@djhohnstein

Who I Am

- Author and contributor of various tools on GitHub
  - Part of the Sharp* wave of tools
- Annoying Persistent Twit on Slack to @tifkin and @its_a_feature_
- Senior consultant at SpecterOps
- Made this tool.
- Claim to fame:
  - Raphael Mudge said my name in a video once.
  - Created a full-length mixtape distributed to my closest friends and family.

Who I Am *Not*

- Computer Science Major
- Professional Software Developer
  - Professional .NET Developer
  - Front-end Designer
- Someone who designs bug-free code

Mythic

Design and Architecture

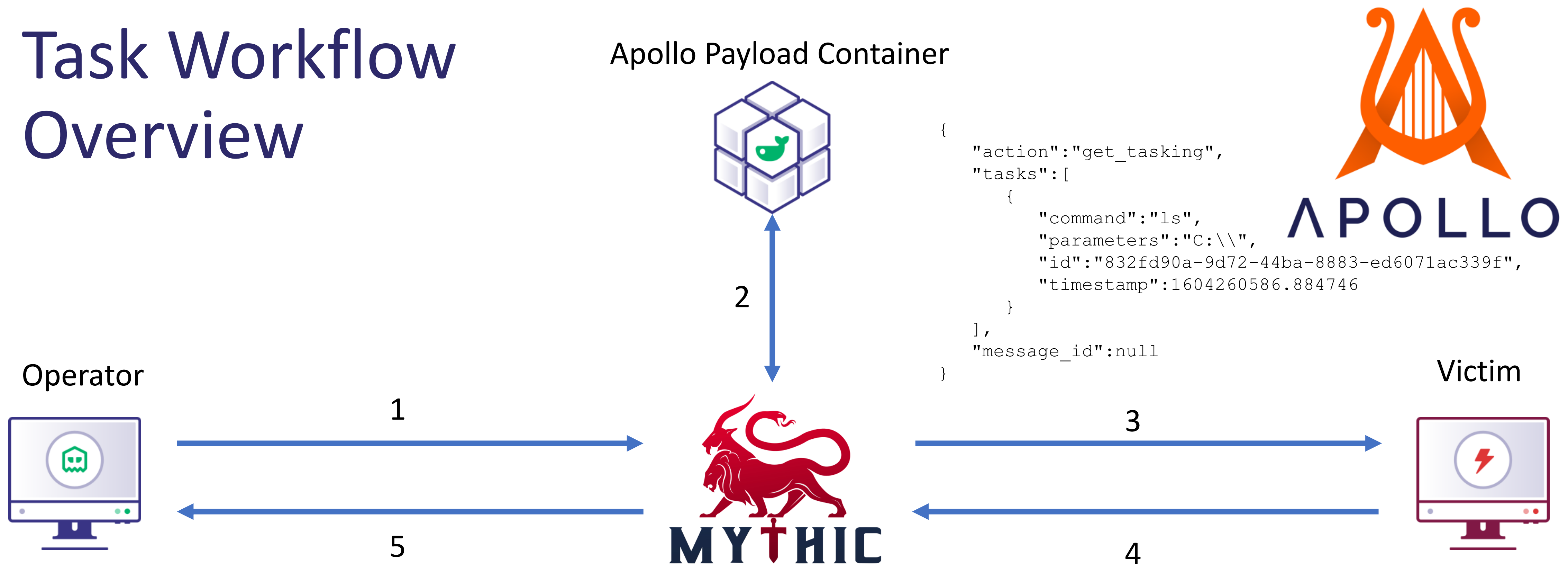
What is Mythic?

- Open-source C2 by @its_a_feature_
- Designed with modularity in-mind
  - Profiles
  - Payloads
  - Taskings
    - Pre and post-processing
    - Customized output
- Easy to use and well-documented API
- Built-in documentation and customizable front-end

What are Payload Containers?

- Docker containers that control payload actions
    - Split into two components:
        - Agent source
        - Mythic interfaces
- Controls how the payload is created in builder.py
- Commands are registered as individual python files
    - Structured JSON from end-to-end
    - Customizable displays of task output via Browser Scripts

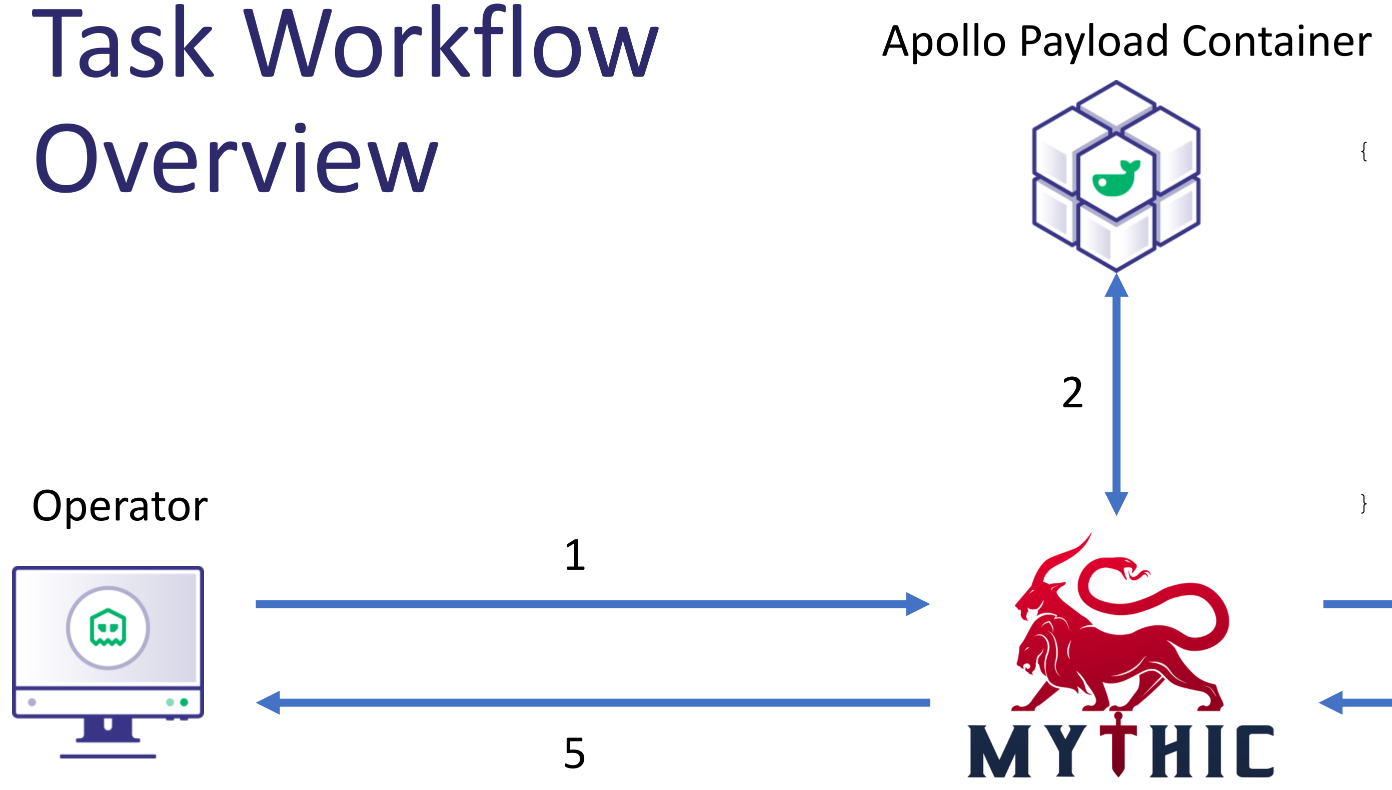
Task Workflow Overview



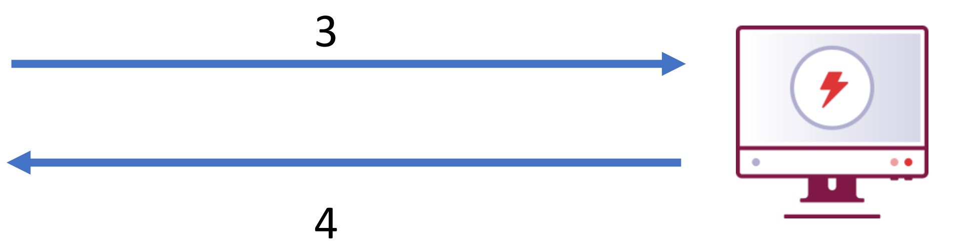
```
{
    "action":"get_tasking",
    "tasks":[
        {
            "command":"ls",
            "parameters":"C:\\",
            "id":"832fd90a-9d72-44ba-8883-ed6071ac339f",
            "timestamp":1604260586.884746
        }
    ],
    "message_id":null
}
```

1. Operator submits task from front-end, like `ls C:\`
2. Payload Container converts command line to JSON blob (top right)
3. Task is sent to the agent
4. Agent performs task and sends results
5. Results are processed by front-end browser (where applicable) before displaying output to user.

Task Workflow Overview



```
{
    "action":"get_tasking",
    "tasks":[
        {
            "command":"ls",
            "parameters":"C:\\",
            "id":"832fd90a-9d72-44ba-8883-ed6071ac339f",
            "timestamp":1604260586.884746
        }
    ],
    "message_id":null
}
```



1. Operator submits task from front-end, like `ls C:\`
2. Payload Container converts command line to JSON blob (top right)
3. Task is sent to the agent
4. Agent performs task and sends results
5. Results are processed by front-end browser (where applicable) before displaying output to user.

What does this allow us to do?

- Pre-Processing on Commands
  - Modal popups and command-lines supported
  - Popups are more user-friendly for new users
  - Ensures data is properly formatted
- Post-Processing on Commands
  - Return basic string or JSON output
  - Browser scripts
    - Highly customizable
    - Draws user attention to what’s important
    - Output sorting and filtering

What is Apollo?

- .NET Framework Agent for Mythic C2
- Designed with Training in Mind
  - Open Source
  - Extensible
  - Transparent Implementation of Attack Techniques
- Leverages Features of Mythic Framework
  - Browser Scripts
  - Wrapper Payloads
  - Lateral Movement
  - Per-operator Task Locking
  - SOCKS
- Usable by Students Before and After Training

Designed for Training?

- Quality of Life Improvements
  - Operator-specific command history + filtering
    - Useful for multiplayer
  - Token impersonation tracking
  - Assembly + Script tracking
  - Verbose command output that's sortable
    - Permissions, paths, clipboard interfaces
- Non-Evasive
  - Lack of built-in evasive egress channels (but extensible)
  - No obfuscation
- Extensible
  - Encourage students to design new features to learn coding in a new way

Notable Features

- Third-Party Tool Execution
  - .NET Assemblies
    - Fork + run, injection
  - PowerShell Scripts
    - Concurrent script imports
    - Loaded script tracking
    - In-process, fork+run, inject
- Automated Payload Compilation
  - Lateral movement (including SMB*)
  - UAC Bypass
- SOCKSv5 Support, End-to-End

Demo:
Building Apollo in Mythic

Commands

- Compiled in via Preprocessor Definitions
- Dispatched via Reflection
- Managed Commands
  - Implemented in agent source
  - ps, ls, shell, etc.
  - No extraneous per-command preprocessing required
- Unmanaged Commands
  - Prebuilt DLLs server-side
  - Leverages Build Pipeline + sRDI within payload container

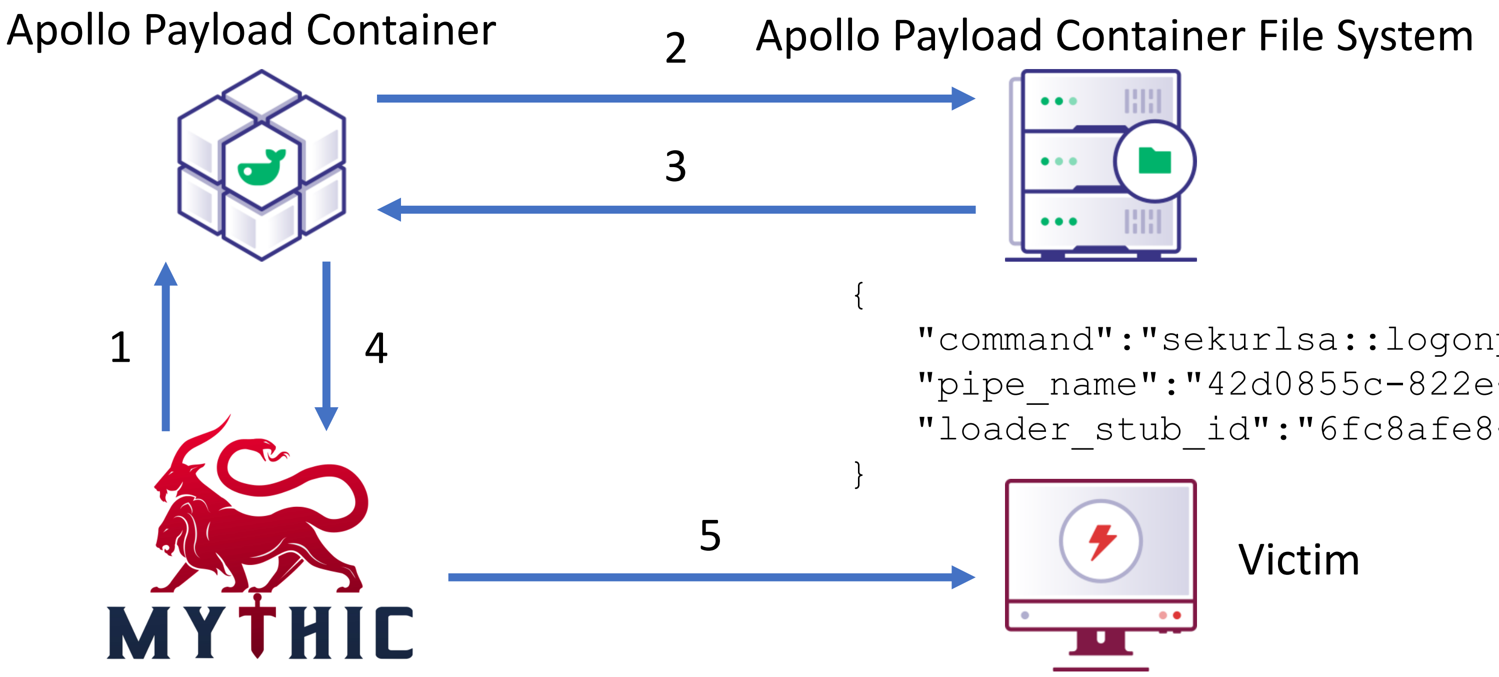
Demo:
Basic Taskings and Quality of Life

Unmanaged Commands

- Projects built as [Command]_[Arch].dll
- On Tasking, Mythic:
  - Finds the architecture appropriate DLL
  - Converts to shellcode using sRDI[1]
  - Retrieves STDIO from Named Pipe until EOF Received
- apollo_smb_server.cpp
  - Added to All Unmanaged Projects
  - Exports Function Without Name Mangling
  - Responsible for Creating Named Pipe Server & Redirecting I/O
- Injection Dynamically Set

1 - https://github.com/monoxgas/sRDI

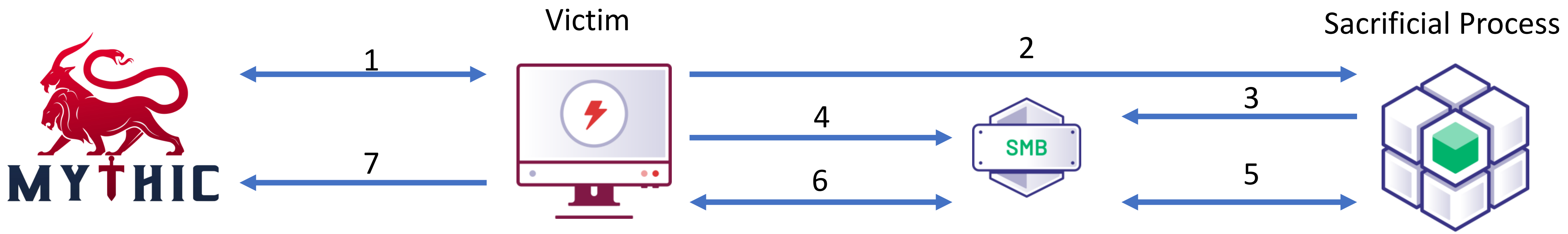
Unmanaged Task Workflow (Server)



```
{
    "command":"sekurlsa::logonpasswords",
    "pipe_name":"42d0855c-822e-4df0-81e5-ee0f80a941c5",
    "loader_stub_id":"6fc8afe8-2b45-4659-af8b-11d78ca67816"
}
```

1. Unmanaged task sent to Apollo container, e.g. mimikatz sekurlsa::logonpasswords
2. Apollo container retrieves Mimikatz_{ARCH}.dll from file system
3. Convert DLL to shellcode via sRDI using exported function requiring pipe name
4. Register shellcode as a file with Mythic, finish populating command parameters
5. Submit task to agent (client)

Unmanaged Task Workflow (Client)



1. On task receive, fetch file specified by loader_stub_id
2. Start process (rundll32.exe) suspended and inject modified Mimikatz shellcode
3. Shellcode in process opens named pipe server specified by pipe_name and waits
4. Apollo connects to Named pipe server, sending specified Mimikatz commands
5. Server reads commands from and writes output to named pipe
6. Apollo reads from named pipe until “EOF”
7. Submit results Mythic

Unmanaged Commands

- mimikatz
- execute_assembly
- assembly_inject
- powerpick
- psinject
- printspoofer
- keylog
- spawn

Demo:
Mimikatz

Demo:
Lateral Movement

Third-Party .NET Assembly Execution

- Based on DotNetReflectiveLoading[2] Project
- Fork & Run or Inject into Remote Process
- No On-Disk Tab Completion
  - Limitation of Web UI
- Broken Out into Three Components
  - Registration
    - register_assembly
  - Execution
    - execute_assembly
    - inject_assembly
  - Unload
    - unload_assembly

2 - https://github.com/ambray/DotNetReflectiveLoading

PowerShell Execution

- Three variants:
    - powershell
        - In-Process Execution
    - psinject, powerpick
        - Execution via sRDI
- Multiple script imports supported
    - Tracks Imported Scripts
    - Cannot Autocomplete Cmdlet Names

Demo:
Third-Party Assembly and PowerShell Workflow

Payload Building and Tracking

- Commands can build new payloads
  - Artifact Tracking
- Wrappers Extend Beyond Agent Deployment
  - Lateral movement
  - Privilege escalation
  - Initial access
- Built-in Lateral Movement to Agent
  - psexec Creates New Service
  - Service Executable Unique Per-Command Issued

Demo:
SOCKSv5 Implementation

Conclusion

- Solid, stable, core functionality
  - All the hits command-wise + SOCKSv5
- Open source allows modification to each component:
  - Task preprocessing
  - Task processing
  - Task output
- Approachable
  - Verbose documentation
- Coming soon to a training near you!

www.specterops.io
@specterops
info@specterops.io