Microsoft has discovered several vulnerabilities, collectively referred to as Nimbuspwn, that could allow an attacker to elevate privileges to root on many Linux desktop endpoints. The vulnerabilities can be chained together to gain root privileges on Linux systems, allowing attackers to deploy payloads, like a root backdoor, and perform other malicious actions via arbitrary root code execution. Moreover, the Nimbuspwn vulnerabilities could potentially be leveraged as a vector for root access by more sophisticated threats, such as malware or ransomware, to achieve greater impact on vulnerable devices.

We discovered the vulnerabilities by listening to messages on the System Bus while performing code reviews and dynamic analysis on services that run as root, noticing an odd pattern in a systemd unit called networkd-dispatcher. Reviewing the code flow for networkd-dispatcher revealed multiple security concerns, including directory traversal, symlink race, and time-of-check-time-of-use race condition issues, which could be leveraged to elevate privileges and deploy malware or carry out other malicious activities. We shared these vulnerabilities with the relevant maintainers through Coordinated Vulnerability Disclosure (CVD) via Microsoft Security Vulnerability Research (MSVR). Fixes for these vulnerabilities, now identified as CVE-2022-29799 and CVE-2022-29800, have been successfully deployed by the maintainer of the networkd-dispatcher, Clayton Craft. We wish to thank Clayton for his professionalism and collaboration in resolving those issues. Users of networkd-dispatcher are encouraged to update their instances.

As organizational environments continue to rely on a diverse range of devices and systems, they require comprehensive solutions that provide cross-platform protection and a holistic view of their security posture to mitigate threats, such as Nimbuspwn. The growing number of vulnerabilities on Linux environments emphasize the need for strong monitoring of the platform's operating system and its components. Microsoft Defender for Endpoint enables organizations to gain this necessary visibility and detect such threats on Linux devices, allowing organizations to detect, manage, respond, and remediate vulnerabilities and threats across different platforms, including Windows, Linux, Mac, iOS, and Android.

In this blog post, we will share some information about the affected components and examine the vulnerabilities we uncovered. Detailing how our cross-domain visibility helps us uncover new and unknown threats to continually improve security, we are also sharing details from our research with the larger security community to underscore the importance of securing platforms and devices.

Background — D-Bus

D-Bus (short for "Desktop-Bus") is an inter-process communication channel (IPC) mechanism developed by the freedesktop.org project. D-Bus is a software-bus and allows processes on the same endpoint to communicate by transmitting messages and responding to them. D-Bus supports two main ways of communicating:

1. Methods — used for request-response communications.
2. Signals — used for publish/subscribe communications.

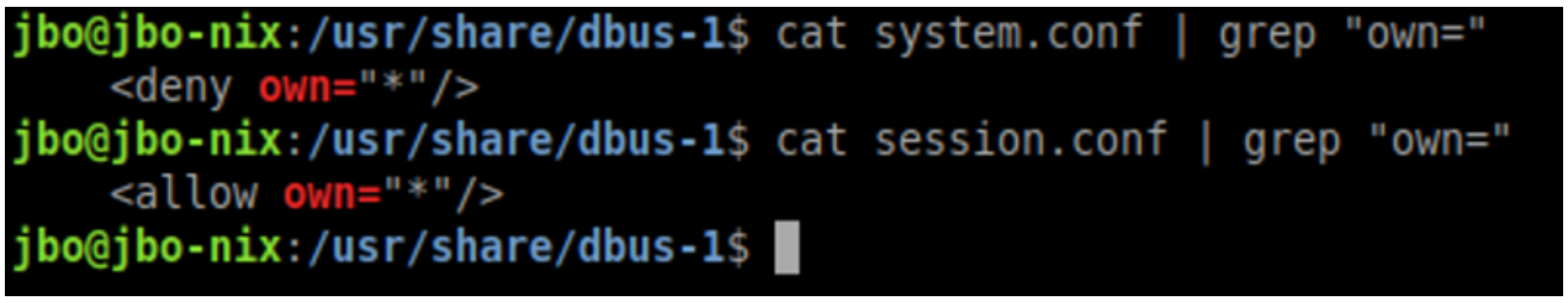
An example of D-Bus usage would be receiving a video chat by a popular video conferencing app—once a video is established, the video conferencing app could send a D-bus signal publishing that a call has started. Apps listening to that message could respond appropriately—for example, mute their audio.

There are many D-Bus components shipped by default on popular Linux desktop environments. Since those components run at different privileges and respond to messages, D-Bus components are an attractive target for attackers. Indeed, there have been interesting vulnerabilities in the past related to buggy D-Bus services, including USBCreator Elevation of Privilege, Blueman Elevation of Privilege by command injection, and other similar scenarios.

D-Bus exposes a global System Bus and a per-session Session Bus. From an attacker's perspective, the System Bus is more attractive since it will commonly have services that run as root listening to it.

D-Bus name ownership

When connecting to the D-Bus, components are assigned with a unique identifier, which mitigates against attacks abusing PID-recycling. The unique identifier starts with a colon and has numbers in it separated by dots, such as ":1.337". Components can use the D-Bus API to own identifiable names such as "org.freedesktop.Avahi" or "com.ubuntu.SystemService". For D-Bus to allow such ownership, the requesting process context must be allowed under the D-Bus configuration files. Those configuration files are well documented and maintained under /usr/local/share/dbus-1/system.conf and /usr/local/share/dbus-1/session.conf (on some systems under /usr/local/dbus-1 directly). Specifically, the default system.conf does not allow ownership unless specified otherwise in other included configuration files (commonly under /etc/dbus-1/system.d).



```
jbo@jbo-nix:/usr/share/dbus-1$ cat system.conf | grep "own="
    <deny own="*"/>
jbo@jbo-nix:/usr/share/dbus-1$ cat session.conf | grep "own="
    <allow own="*"/>
jbo@jbo-nix:/usr/share/dbus-1$ 
```

Figure 1: Different ownership policies for the System Bus and the Session Bus

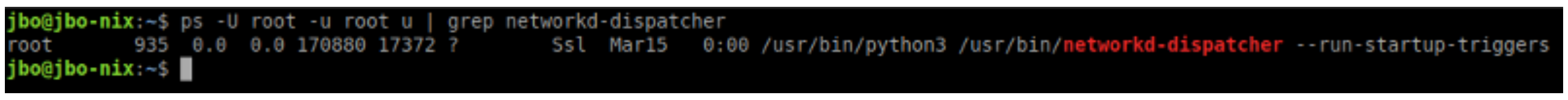
Additionally, if the name requested already exists—the request will not be granted until the owning process releases the name.

Vulnerability hunting

Our team has started enumerating services that run as root and listen to messages on the System Bus, performing both code reviews and dynamic analysis. We have reported two information leak issues as a result:

1. Directory Info Disclosure in Blueman
2. Directory Info Disclosure in PackageKit (CVE-2022-0987)

While these are interesting, their severity is low — an attacker can list files under directories that require high permissions to list files under. Then we started noticing interesting patterns in a systemd unit called networkd-dispatcher. The goal of networkd-dispatcher is to dispatch network status changes and optionally perform different scripts based on the new status. Interestingly, it runs on boot as root:



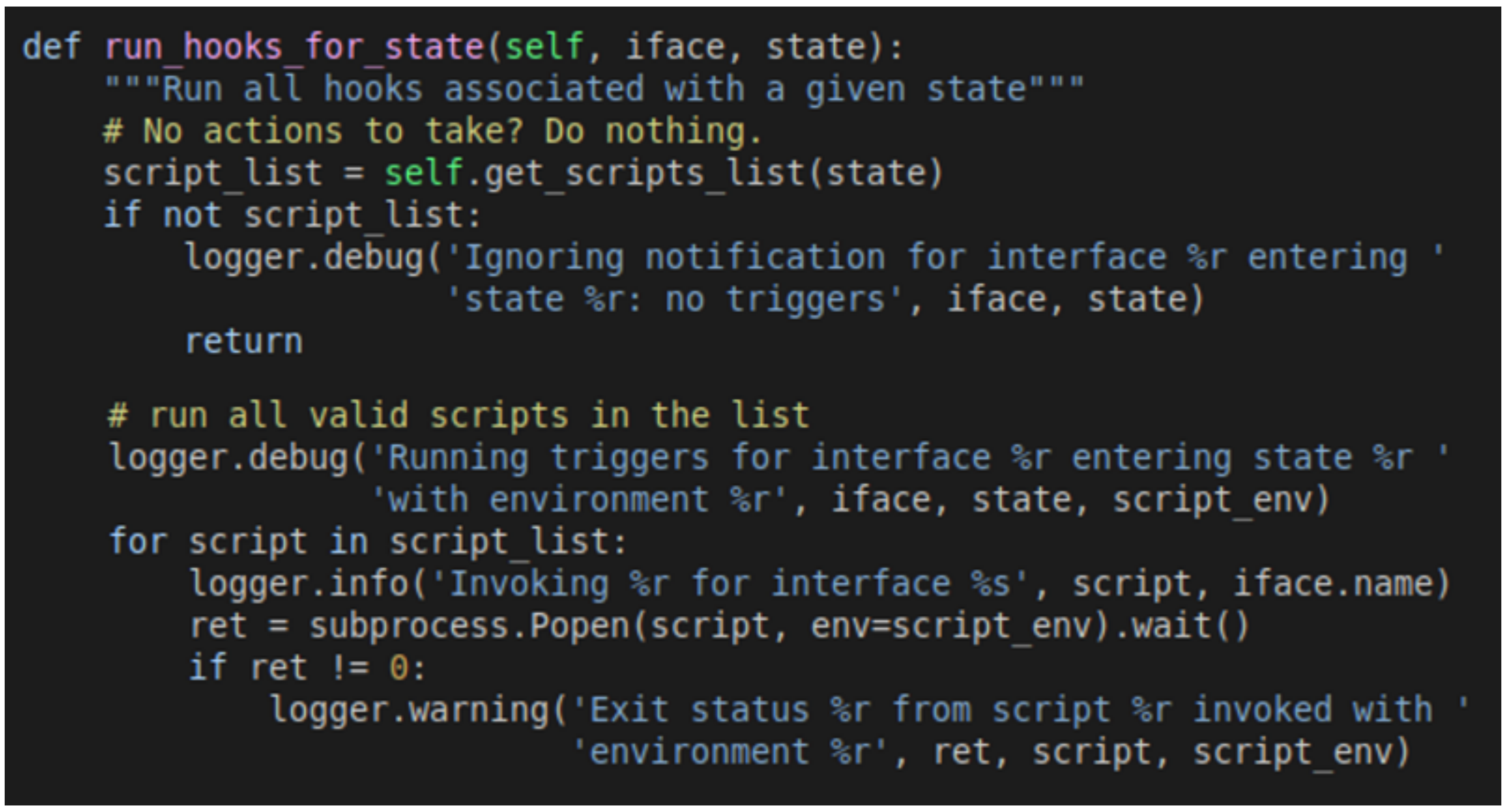
```
jbo@jbo-nix:~$ ps -U root -u root u | grep networkd-dispatcher
root         935  0.0  0.0 170880 17372 ?        Ssl  Mar15   0:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
jbo@jbo-nix:~$ 
```

Figure 2: networkd-dispatcher running as root

Code flow for networkd-dispatcher

Upon examination of the networkd-dispatcher source code, we noticed an interesting flow:

1. The register function registers a new signal receiver for the service "org.freedesktop.network1" on the System Bus, for the signal name "PropertiesChanged".
2. The "_receive_signal" signal handler will perform some basic checks on the object type being sent, concludes the changed network interface based on the object path being sent, and then concludes its new states—"OperationalState" and "AdministrativeState"—each fetched from the data. For any of those states—if they aren't empty—the "handle_state" method will get invoked.
3. The "handle_state" method simply invokes "_handle_one_state" for each of those two states.
4. "_handle_one_state" validates the state isn't empty and checks if it's different than the previous state. If it is, it will update the new state and invoke the "_run_hooks_for_state" method, which is responsible of discovering and running the scripts for the new state.
5. "_run_hooks_for_state" implements the following logic:
   - Discovers the script list by invoking the "get_script_list" method (which gets the new state as a string). This method simply calls "scripts_in_path" which is intended to return all the files under "/etc/networkd-dispatcher/<state>.d" that are owned by the root user and the root group, and are executable.
   - Sorts the script list.
   - Runs each script with subprocess.Popen while supplying custom environment variables.



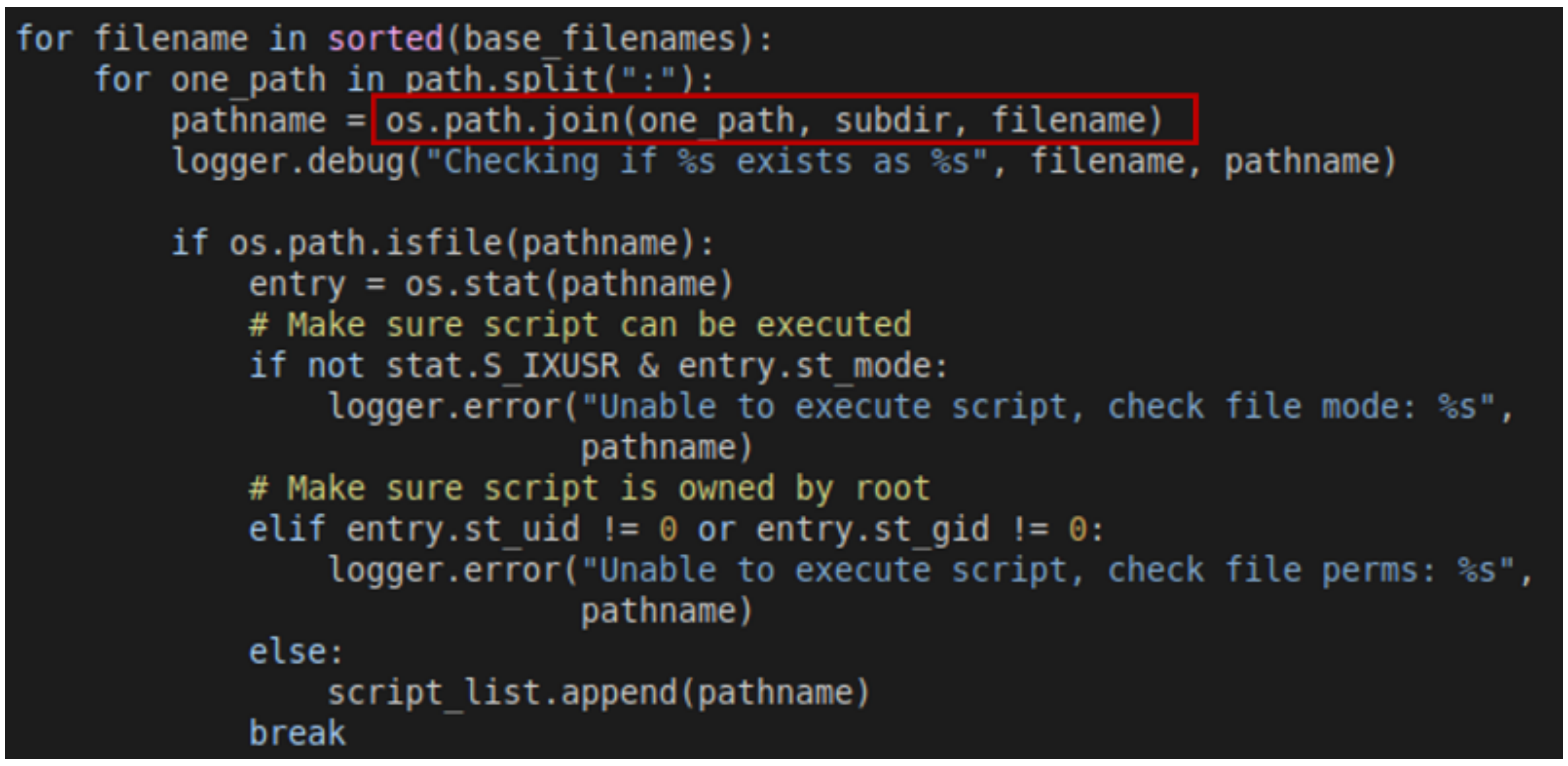
```python
def run_hooks_for_state(self, iface, state):
    """Run all hooks associated with a given state"""
    # No actions to take? Do nothing.
    script_list = self.get_scripts_list(state)
    if not script_list:
        logger.debug('Ignoring notification for interface %r entering '
                     'state %r: no triggers', iface, state)
        return

    # run all valid scripts in the list
    logger.debug('Running triggers for interface %r entering state %r '
                 'with environment %r', iface, state, script_env)
    for script in script_list:
        logger.info('Invoking %r for interface %s', script, iface.name)
        ret = subprocess.Popen(script, env=script_env).wait()
        if ret != 0:
            logger.warning('Exit status %r from script %r invoked with '
                           'environment %r', ret, script, script_env)
```

Figure 3: _run_hooks_for_state source code — some parts omitted for brevity

Step 5 has multiple security issues:

1. Directory traversal (CVE-2022-29799): none of the functions in the flow sanitize the OperationalState or the AdministrativeState. Since the states are used to build the script path, it is possible that a state would contain directory traversal patterns (e.g. "../../") to escape from the "/etc/networkd-dispatcher" base directory.
2. Symlink race: both the script discovery and subprocess.Popen follow symbolic links.
3. Time-of-check-time-of-use (TOCTOU) race condition (CVE-2022-29800): there is a certain time between the scripts being discovered and them being run. An attacker can abuse this vulnerability to replace scripts that networkd-dispatcher believes to be owned by root to ones that are not.



```python
for filename in sorted(base_filenames):
    for one_path in path.split(":"):
        pathname = os.path.join(one_path, subdir, filename)
        logger.debug("Checking if %s exists as %s", filename, pathname)

        if os.path.isfile(pathname):
            entry = os.stat(pathname)
            # Make sure script can be executed
            if not stat.S_IXUSR & entry.st_mode:
                logger.error("Unable to execute script, check file mode: %s",
                             pathname)
            # Make sure script is owned by root
            elif entry.st_uid != 0 or entry.st_gid != 0:
                logger.error("Unable to execute script, check file perms: %s",
                             pathname)
            else:
                script_list.append(pathname)
            break
```

Figure 4: Building the script list in the "scripts_in_path" method, including the vulnerable code with "subdir" poisoned.

Exploitation

Let us assume an adversary has a malicious D-Bus component that can send an arbitrary signal. An attacker can therefore do the following:

1. Prepare a directory "/tmp/nimbuspwn" and plant a symlink "/tmp/nimbuspwn/poc.d" to point to "/sbin". The "/sbin" directory was chosen specifically because it has many executables owned by root that do not block if run without additional arguments. This will abuse the symlink race issue we mentioned earlier.
2. For every executable filename under "/sbin" owned by root, plant the same filename under "/tmp/nimbuspwn". For example, if "/sbin/vgs" is executable and owned by root, plant an executable file "/tmp/nimbuspwn/vgs" with the desired payload. This will help the attacker win the race condition imposed by the TOCTOU vulnerability.
3. Send a signal with the OperationalState "../../../tmp/nimbuspwn/poc". This abuses the directory traversal vulnerability and escapes the script directory.

4. The networkd-dispatcher signal handler kicks in and builds the script list from the directory “/etc/networkd-dispatcher/../../../tmp/nimbuspwn/poc.d”, which is really the symlink (“/tmp/nimbuspwn/poc.d”), which points to “/sbin”. Therefore, it creates a list composed of many executables owned by root.
5. Quickly change the symlink “/tmp/nimbuspwn/poc.d” to point to “/tmp/nimbuspwn”. This abuses the TOCTOU race condition vulnerability—the script path changes without networkd-dispatcher being aware.
6. The dispatcher starts running files that were initially under “/sbin” but in truth under the “/tmp/nimbuspwn” directory. Since the dispatcher “believes” those files are owned by root, it executes them blindly with subprocess.Popen as root. Therefore, our attacker has successfully exploited the vulnerability.

Note that to win the TOCTOU race condition with high probability, we plant many files that can potentially run. Our experiments show three attempts were enough to win the TOCTOU race condition.

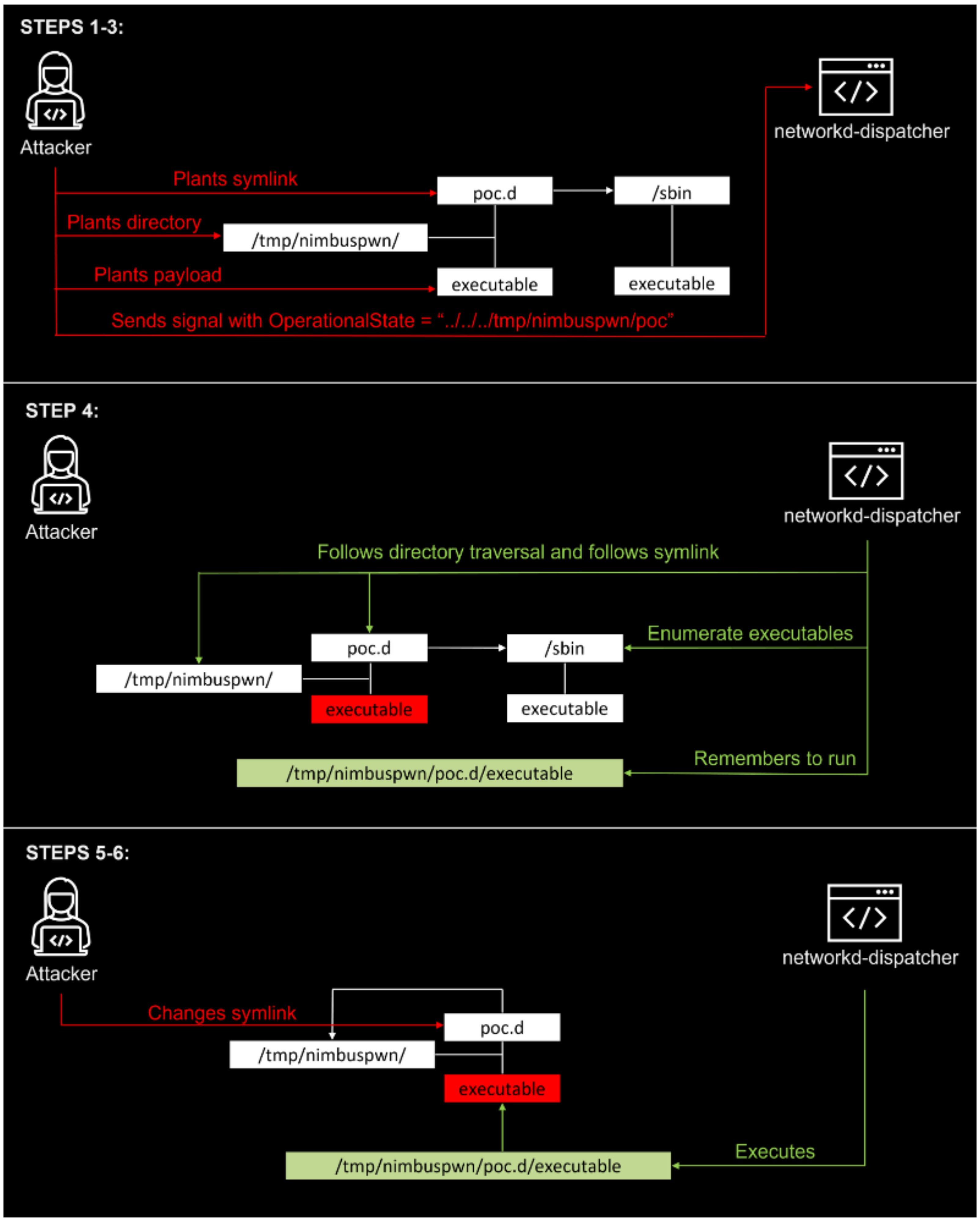


Figure 5: Flow-chart of the attack in three stages

Since we do not wish to run the exploit every time we want to run as root, the payload that we ended up implementing leaves a root backdoor as such:

1. Copies /bin/sh to /tmp/sh.
2. Turns the new /tmp/sh it into a Set-UID (SUID) binary.
3. Run /tmp/sh -p. The “-p” flag is necessary since modern shells drop privileges by design.

Owning the bus name

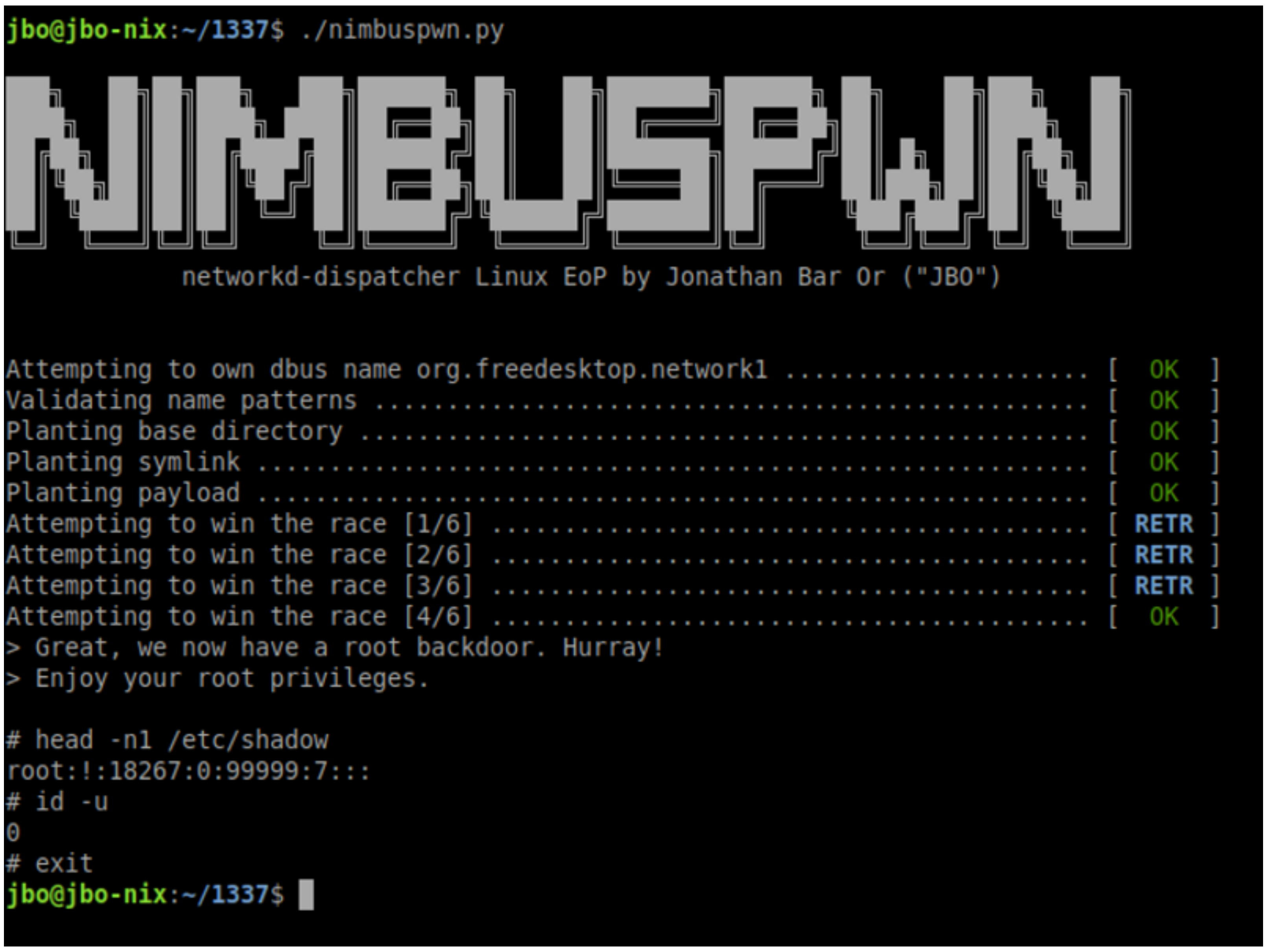
The astute reader will notice that the entire exploit elevates privileges assuming our exploit code can own the "org.freedesktop.network1" bus name. While this sounds non-trivial, we have found several environments where this happens. Specifically:

1. On many environments (e.g. Linux Mint) the service systemd-networkd that normally owns the "org.freedesktop.network1" bus name does not start at boot by default.
2. Using advanced hunting in Microsoft Defender for Endpoint we were able to spot several processes running as the systemd-network user (which is permitted to own the bus name we require) running arbitrary code from world-writable locations. These include several gpgv plugins (launched when apt-get installs or upgrades) as well as the Erlang Port Mapper Daemon (epmd) which allows running arbitrary code under some scenarios.

The query we used can also be run by Microsoft Defender for Endpoint customers:

DeviceProcessEvents | where Timestamp > ago(5d) and AccountName == "systemd-network" and isnotempty(InitiatingProcessAccountName) and isnotempty(FileName) | project DeviceId, FileName, FolderPath, ProcessCommandLine

We were therefore able to exploit these scenarios and implement our own exploit:



```
jbo@jbo-nix:~/1337$ ./nimbuspwn.py
NIMBUSPWN
         networkd-dispatcher Linux EoP by Jonathan Bar Or ("JBO")

Attempting to own dbus name org.freedesktop.network1 ..................... [  OK  ]
Validating name patterns ................................................. [  OK  ]
Planting base directory .................................................. [  OK  ]
Planting symlink ......................................................... [  OK  ]
Planting payload ......................................................... [  OK  ]
Attempting to win the race [1/6] ......................................... [ RETR ]
Attempting to win the race [2/6] ......................................... [ RETR ]
Attempting to win the race [3/6] ......................................... [ RETR ]
Attempting to win the race [4/6] ......................................... [  OK  ]
> Great, we now have a root backdoor. Hurray!
> Enjoy your root privileges.

# head -n1 /etc/shadow
root:!:18267:0:99999:7:::
# id -u
0
# exit
jbo@jbo-nix:~/1337$
```

Figure 6: Our exploit implemented and winning the TOCTOU race

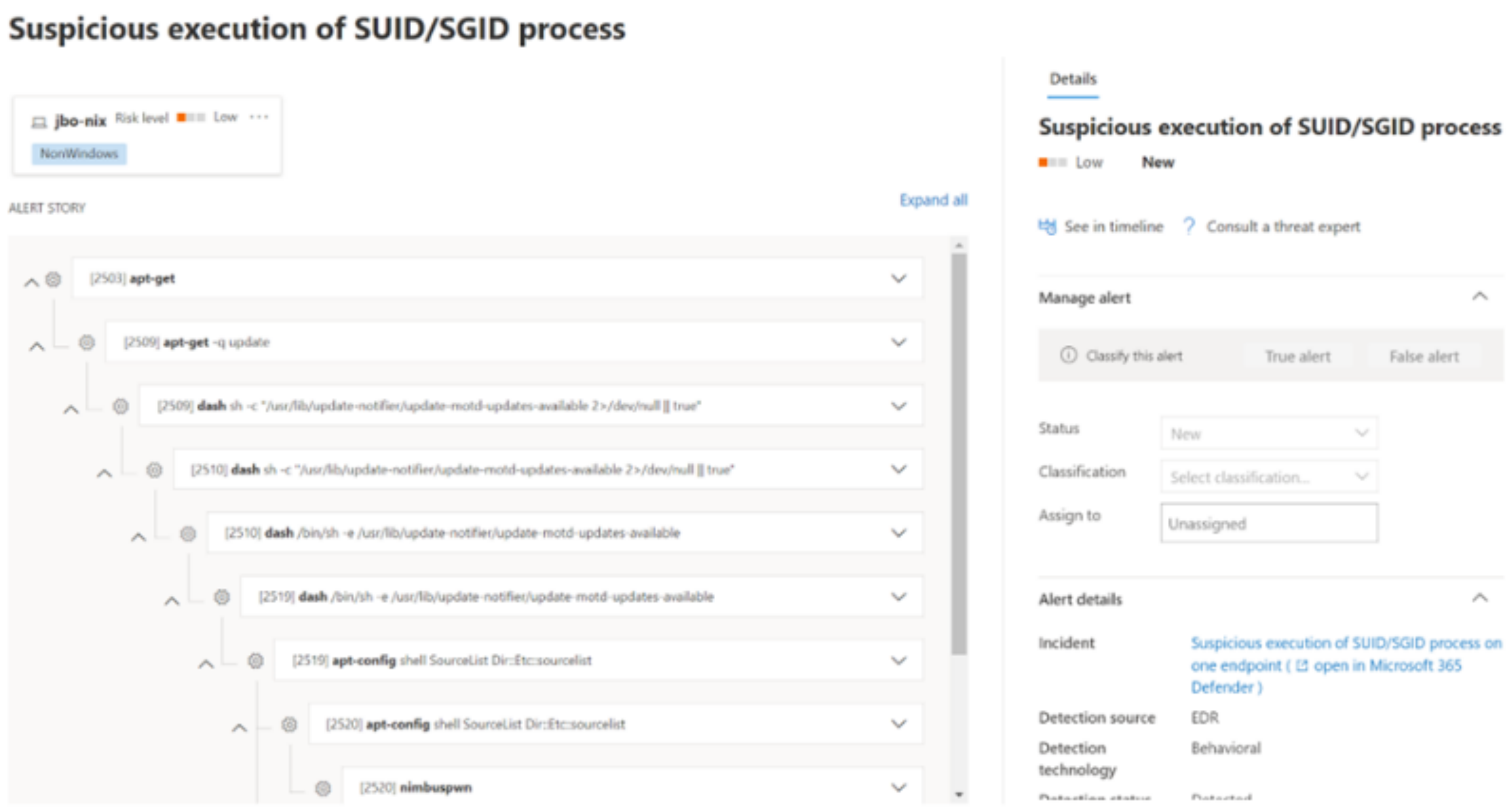
While capable of running any arbitrary script as root, our exploit copies /bin/sh to the /tmp directory, sets /tmp/sh as a Set-UID (SUID) executable, and then invokes "/tmp/sh -p". Note that the "-p" flag is necessary to force the shell to not drop privileges.

Hardening device security and detection strategy

Despite the evolving threat landscape regularly delivering new threats, techniques, and attack capabilities, adversaries continue to focus on identifying and taking advantage of unpatched vulnerabilities and misconfigurations as a vector to access systems, networks, and sensitive information for malicious purposes. This constant bombardment of attacks spanning a wide range of platforms, devices, and other domains emphasizes the need for a comprehensive and proactive vulnerability management approach that can further identify and mitigate even previously unknown exploits and issues.

Microsoft's threat and vulnerability management capabilities help organizations monitor their overall security posture, providing real-time insights into risk with continuous vulnerability discovery, contextualized intelligent prioritization, and seamless one-click flaw remediation. Leveraging our research into the Nimbuspwn vulnerabilities to improve solutions, our threat and vulnerability management already covers CVE-2022-29799 and CVE-2022-29800 and indicates such vulnerable devices in the threat and vulnerability module in Microsoft Defender for Endpoint.

To address the specific vulnerabilities at play, Microsoft Defender for Endpoint's endpoint detection and response (EDR) capabilities detect the directory traversal attack required to leverage Nimbuspwn. Additionally, the Microsoft Defender for Endpoint detection team has a generic detection for suspicious Set-UID process invocations, which detected our exploit without prior knowledge.



Suspicious execution of SUID/SGID process

jbo-nix Risk level Low

NonWindows

ALERT STORY

```
[2503] apt-get
  [2509] apt-get -q update
    [2509] dash sh -c "/usr/lib/update-notifier/update-motd-updates-available 2>/dev/null || true"
      [2510] dash sh -c "/usr/lib/update-notifier/update-motd-updates-available 2>/dev/null || true"
        [2510] dash /bin/sh -e /usr/lib/update-notifier/update-motd-updates-available
          [2519] dash /bin/sh -e /usr/lib/update-notifier/update-motd-updates-available
            [2519] apt-config shell SourceList Dir::Etc::sourcelist
              [2520] apt-config shell SourceList Dir::Etc::sourcelist
                [2520] nimbuspwn
```

Details

Suspicious execution of SUID/SGID process

Low New

Manage alert

Status	New
Classification	Select classification...
Assign to	Unassigned

Alert details

Incident	Suspicious execution of SUID/SGID process on one endpoint (open in Microsoft 365 Defender)
Detection source	EDR
Detection technology	Behavioral

Figure 7: Microsoft Defender for Endpoint detecting a suspicious SUID process used in our exploit

Defending against the evolving threat landscape requires the ability to protect and secure users' computing experiences, be it a Windows or non-Windows device. Microsoft continuously enriches our protection technologies through robust research that protects users and organizations across all the major platforms every single day. This case displayed how the ability to coordinate such research via expert, cross-industry collaboration is vital to effectively mitigate issues, regardless of the vulnerable device or platform in use. By sharing our research and other forms of threat intelligence, we can continue to collaborate with the larger security community and strive to build better protection for all.

Jonathan Bar Or

Microsoft 365 Defender Research Team