HONEYCOMB

Imri Goldberg @lorgandon - Cymmetria

Omer Cohen @omercnet - Demisto

https://github.com/Cymmetria/honeycomb

Agenda

- Writing honeypots: Goals
- Honeypot types
- What makes a honeypot good and useful
- Step-by-step guide to writing a honeypot
- About Honeycomb and how it can help you

Honeypot ecosystem

- There are a lot of honeypots today: https://github.com/paralax/awesome-honeypots
- Most honeypots are “non-standard”:
  - Specific to a particular protocol
  - Sometimes configurable in their own way
  - Reimplement reporting
  - Reimplement execution/deployment

What is Honeycomb?

- **A library for honeypots**
- Easy to:
  - Write new honeypots
  - Gain exposure for new honeypots
  - Get and run others’ honeypots
- Provide infrastructure:
  - Installation
  - Configuration of honeypot parameters, e.g. files in FTP, banner for Telnet, etc.
  - Reporting, e.g. syslog, MISP, …
  - Supports low- and medium-interaction honeypots easily
  - Supports high-interaction honeypots with Docker images (e.g. Struts, MongoDB)

Honeycomb example usage

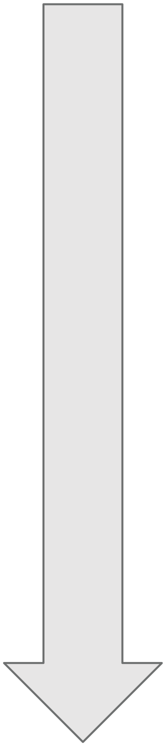
```
# get the list of available honeypots
$ honeycomb service list -r

# install a honeypot, in its own virtualenv
$ honeycomb service install hp_officejet

# configure syslog
$ honeycomb integration configure syslog protocol=udp
address=127.0.0.1 port=5555

# run the honeypot with syslog configured
$ honeycomb service run hp_officejet -i syslog
```

Why should we write honeypots anyway?

- Understanding which attacks are prevalent today
  - Capturing attackers’ tools - e.g. the attacker tries to upload her malware to our honeypot
  - Mapping origin IP addresses - e.g. all attacks are coming from 1.2.3.4
  - Discovering and fingerprinting the vulnerabilities attackers are using
  - Understanding which attacks are targeting a specific protocol, product, vulnerability
- Knowing what is targeting me
  - Create a honeypot that’s somehow related to your organization and see what’s hitting it
    - Using dark web breadcrumbs
    - Using your own IP space or DNS addresses
    - Using deceptive personas
- Catching attackers on my network

Types of honeypots

- “Fake script” (in Python or another language…)
  - Just listen on the appropriate port
  - Provide the correct banner
  - Interact correctly for the first few steps of the protocol
  - Full protocol implementation
  - Simulate the underlying OS - e.g. allow to upload files
- Actual software, running on a docker/LXC image
  - e.g. MongoDB honeypot
- Actual software running on a VM



Low interaction to high interaction

(Degree of realism)

What makes a honeypot good and useful

- First and foremost: Does it satisfy the goals we built it for?
- More generally:
  - Operations:
    - Can it report to external collectors? Specifically: Can it be integrated with existing threat intelligence platforms?
    - Can it be deployed easily?
  - Deception:
    - Does it fool an automated scanner for a specific exploit?
    - Does it fool any generic automated scanner for the service?
    - Does it fool a human using this?

Actually writing very real honeypots

Cymmetria

Things every honeypot needs

- A mechanism for emitting alerts
  - Such a mechanism should allow reporting a timestamp, source address, event description, event severity (or priority)
  - It should be flexible enough to allow adding other fields (e.g. command, filename, URL, path, etc.)
- A way to start up, shut it down
- An easy way to install it
  - For a Python script, either a setup.py or at least requirements.txt in pip install format
- When using Honeycomb you get all of the above

What do you need before starting

- In order for your honeypot to be believable, you need to research thoroughly how the system actually looks and feels
    - If you’re writing a honeypot for service X, it’s good to have the actual real service available to you
        - Example: if you’re writing a Cisco Telnet honeypot, it’s good to have a Cisco router available with Telnet open
    - It’s important to have a legitimate client for the service if relevant, e.g. a Telnet client
    - If you’re writing a honeypot for a specific exploit:
        - You need the exploitable server
        - You really really want the exploit code available to you

Collecting intelligence before writing a honeypot

- Where can I get a real server for protocol/product X?
  - If it’s open source, just try to run it
  - Borrow hardware from a friend
  - We have friends who find them online, using tools such as Shodan. They may search for an indicative string, even the name of the product itself.
- Where can I get the exploit?
  - Exploits are published on Metasploit, Exploit-DB, etc.
  - Usually the CVE database will have a link to the exploit

How to write a generic low-interaction honeypot

- There are two possible directions:
  - Use an existing library that implements the protocol, or implement from scratch
- When using an existing library (the easier choice):
  - Implement the server
  - Compare to a reference server
  - Interact with it with a client
  - If you are looking to fool a human, the UI (e.g. appearance in a browser) needs to look good
  - If you are looking to fool a machine, the protocol needs to work well. If it’s a website, the source must look right and match the signatures you expect attacker scripts might be looking for.

How to write a generic low-interaction honeypot (cont’d)

- When writing from scratch:
  - The general approach is similar, but you need to start with some kind of a socket server
  - Use existing libraries for server whenever possible. To keep things simple, a Python SocketServer is good enough.
  - Apart from the reference server, also have the protocol spec ready

How to write a low-interaction honeypot for a specific exploit

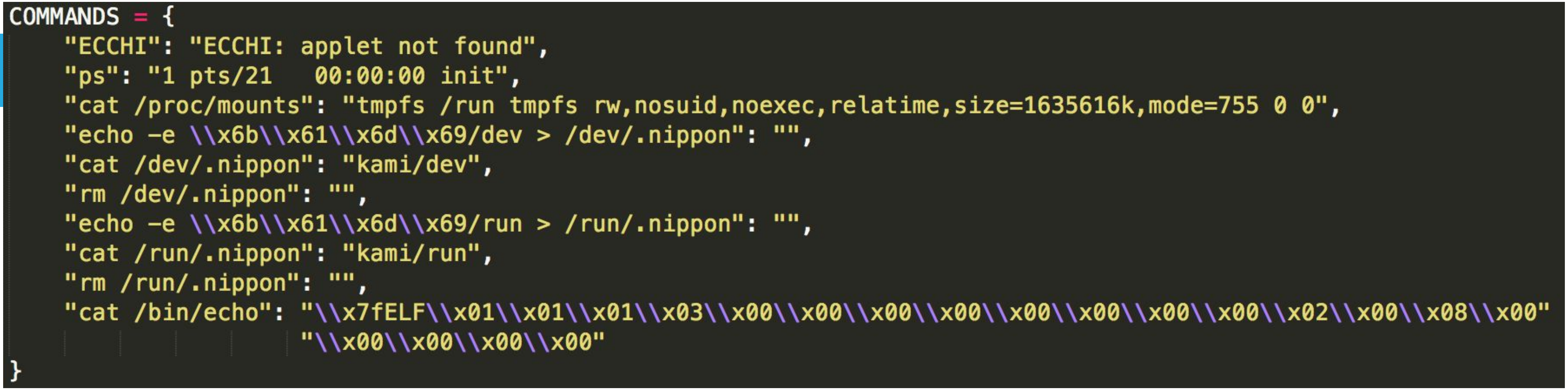
- Start with a generic low-interaction honeypot of any kind
- Test the exploit against it - Does the exploit report success? If no, continue working on the honeypot until it does.
  - Sometimes the exploit will report success only after executing code that beacons back. In that case it is probably not practical to catch this particular behavior…
- In the code of the honeypot, try to differentiate between regular interaction with the honeypot, and detecting the use of the exploit
  - Ideally: you can detect actual usage of a vulnerability
  - Less ideal: you can detect behavior that is specific to the exploit, but not necessarily indicative of it

Example: Mirai worm honeypot

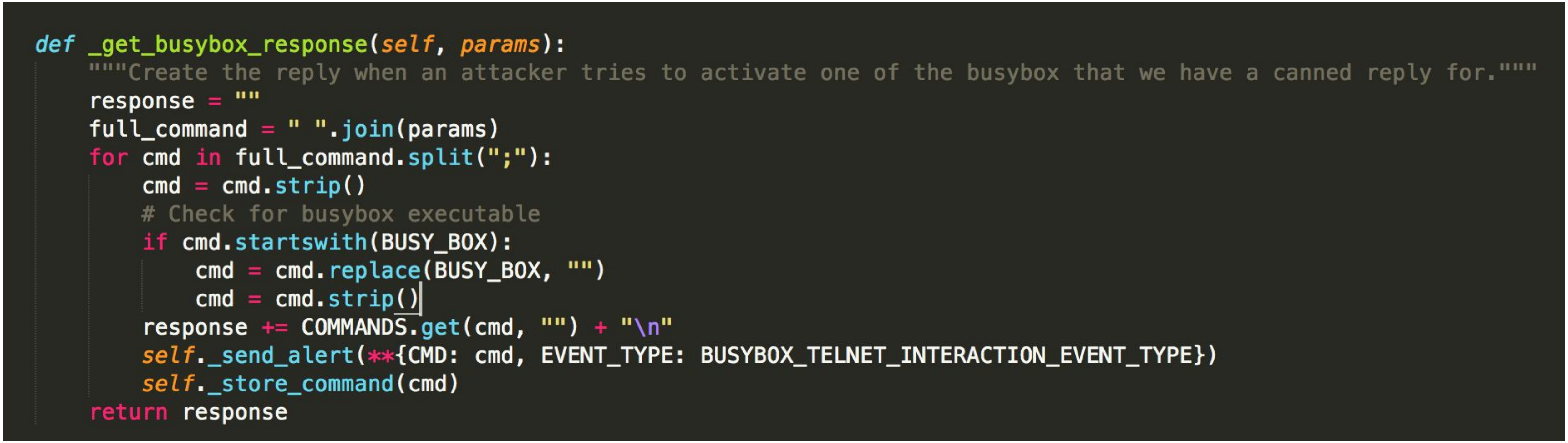
- Mirai worm - spread by infecting security cameras running vulnerable Telnet
- The code had a specific sequence of steps performed on a target:
  - Verify Telnet is open
  - Verify that the security camera is vulnerable
  - Exploit
- Our goal was to create a honeypot that would:
  - Report machines compromised by Mirai (used as attack sources)
  - Catch the executable used to (attempt to) infect our machine
- We had the code of Mirai available

Example: Mirai worm honeypot (continued)

- Our approach:
  - Start with a basic Telnet honeypot that we wrote in Python, based on the open source telnetsrv
  - Patch the Mirai code to attack only that, and run in a controlled environment
  - Learn what the Mirai worm is sending to us, and add code to our honeypot that will check that it is sent
  - Review the Mirai worm code, and understand what it is expecting to get as a response - add code in the honeypot to send it
  - Test the honeypot and see that we get to the next stage
  - Continue until we get to a stage with a definite indication of Mirai attack

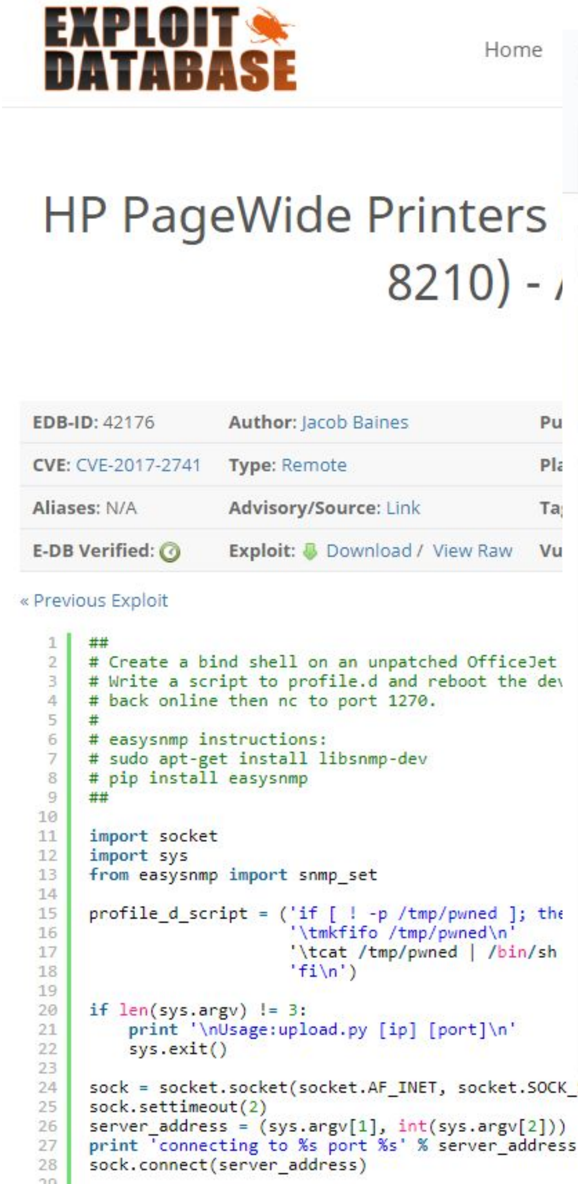


```
COMMANDS = {
    "ECCHI": "ECCHI: applet not found",
    "ps": "1 pts/21   00:00:00 init",
    "cat /proc/mounts": "tmpfs /run tmpfs rw,nosuid,noexec,relatime,size=1635616k,mode=755 0 0",
    "echo -e \\x6b\\x61\\x6d\\x69/dev > /dev/.nippon": "",
    "cat /dev/.nippon": "kami/dev",
    "rm /dev/.nippon": "",
    "echo -e \\x6b\\x61\\x6d\\x69/run > /run/.nippon": "",
    "cat /run/.nippon": "kami/run",
    "rm /run/.nippon": "",
    "cat /bin/echo": "\\x7fELF\\x01\\x01\\x01\\x03\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x02\\x00\\x08\\x00"
                     "\\x00\\x00\\x00\\x00"
}
```



```
def _get_busybox_response(self, params):
    """Create the reply when an attacker tries to activate one of the busybox that we have a canned reply for."""
    response = ""
    full_command = " ".join(params)
    for cmd in full_command.split(";"):
        cmd = cmd.strip()
        # Check for busybox executable
        if cmd.startswith(BUSY_BOX):
            cmd = cmd.replace(BUSY_BOX, "")
            cmd = cmd.strip()
        response += COMMANDS.get(cmd, "") + "\n"
        self._send_alert(**{CMD: cmd, EVENT_TYPE: BUSYBOX_TELNET_INTERACTION_EVENT_TYPE})
        self._store_command(cmd)
    return response
```

Example - HP OfficeJet Pro Printers - CVE-2017-2741

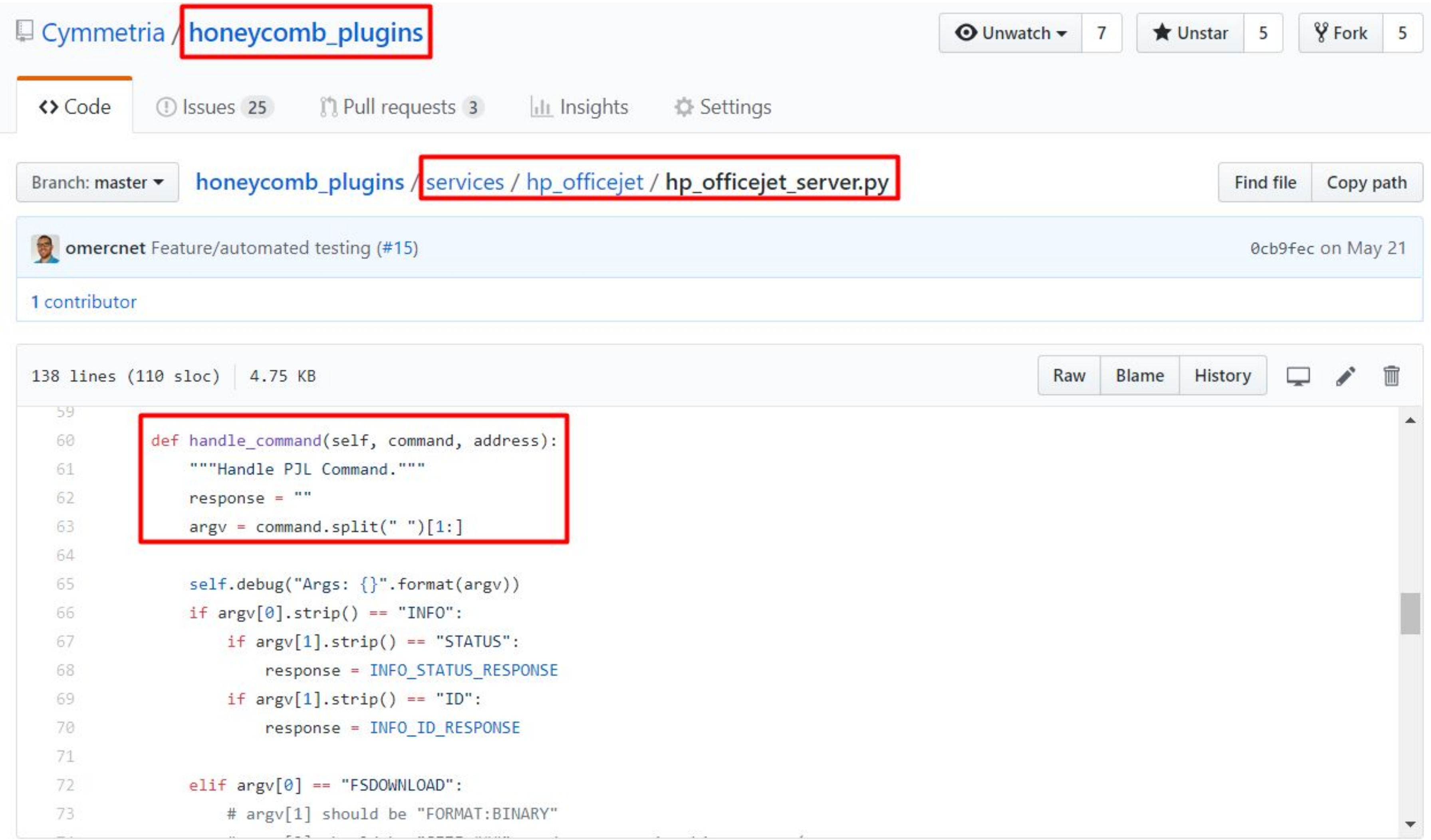


HP PageWide Printers
8210) - /

EDB-ID: 42176	**Author:** Jacob Baines	**Pu**
CVE: CVE-2017-2741	**Type:** Remote	**Pla**
Aliases: N/A	**Advisory/Source:** Link	**Ta**
E-DB Verified:	**Exploit:** Download / View Raw	**Vu**

« Previous Exploit

```
##
# Create a bind shell on an unpatched OfficeJet
# Write a script to profile.d and reboot the dev
# back online then nc to port 1270.
#
# easysnmp instructions:
# sudo apt-get install libsnmp-dev
# pip install easysnmp
##

import socket
import sys
from easysnmp import snmp_set

profile_d_script = ('if [ ! -p /tmp/pwned ]; the
                    '\tmkfifo /tmp/pwned\n'
                    '\tcat /tmp/pwned | /bin/sh
                    'fi\n')

if len(sys.argv) != 3:
    print '\nUsage:upload.py [ip] [port]\n'
    sys.exit()

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.settimeout(2)
server_address = (sys.argv[1], int(sys.argv[2]))
print 'connecting to %s port %s' % server_address
sock.connect(server_address)
```



honeycomb_plugins / services / hp_officejet / hp_officejet_server.py

omercnet Feature/automated testing (#15) 0cb9fec on May 21

1 contributor

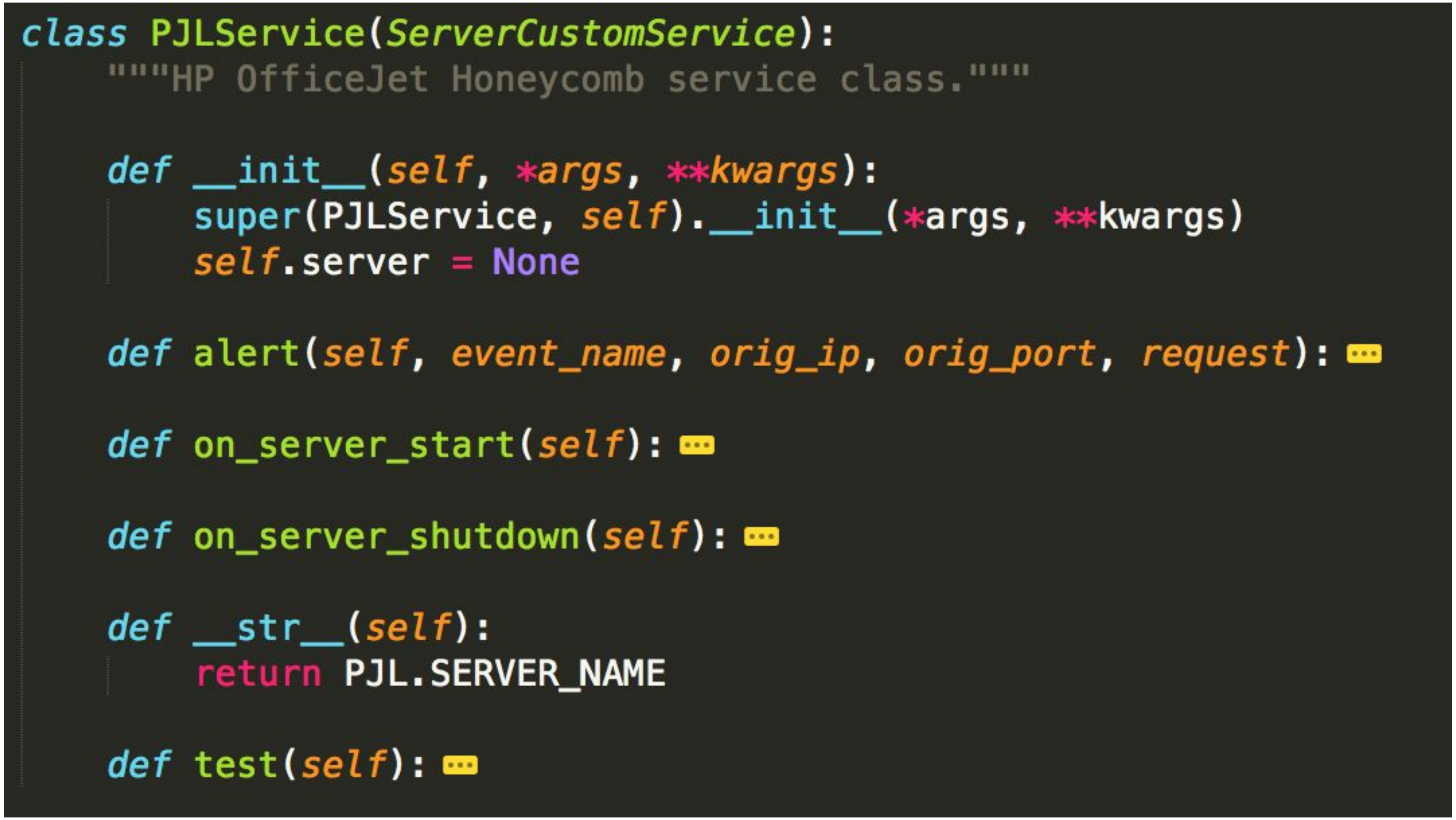
138 lines (110 sloc) 4.75 KB

```
    def handle_command(self, command, address):
        """Handle PJL Command."""
        response = ""
        argv = command.split(" ")[1:]

        self.debug("Args: {}".format(argv))
        if argv[0].strip() == "INFO":
            if argv[1].strip() == "STATUS":
                response = INFO_STATUS_RESPONSE
            if argv[1].strip() == "ID":
                response = INFO_ID_RESPONSE

        elif argv[0] == "FSDOWNLOAD":
            # argv[1] should be "FORMAT:BINARY"
```

How to write a high-interaction honeypot

- Implement standard way to run the actual server
    - Example: MongoDB in a Docker
- Automate startup and shutdown
- Add code to your Docker that will monitor the logs of the application. If you see an interaction event (e.g. successful login, database query, command execution in a shell) emit a high-severity alert.
- For low-importance events (login attempt), emit a low-severity alert
- Example:
    - Struts honeypot: https://github.com/Cymmetria/honeycomb_plugins/blob/feature/struts_service/services/struts/struts_service.py

Honeypot code structure



```
class PJLService(ServerCustomService):
    """HP OfficeJet Honeycomb service class."""

    def __init__(self, *args, **kwargs):
        super(PJLService, self).__init__(*args, **kwargs)
        self.server = None

    def alert(self, event_name, orig_ip, orig_port, request): ⋯

    def on_server_start(self): ⋯

    def on_server_shutdown(self): ⋯

    def __str__(self):
        return PJL.SERVER_NAME

    def test(self): ⋯
```

The most recent example: LibSSH

- CVE-2018-10933 - authentication bypass in LibSSH
- Plenty of exploits available, e.g. https://github.com/SoledaD208/CVE-2018-10933
- Our approach:
  - Create an SSH server using paramiko
  - Patch the server to detect the vulnerability
  - Patch the server to look like LibSSH to scanners

Honeycomb demo

1) https://www.youtube.com/watch?v=F3932X-mhto

2) Live LibSSH demo