International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

Securing AWS EC2: Streamlining IMDS Transition from Third-Party IMDSv1 Calls to IMDSv2 with Proxy Server Integration

Balasubrahmanya Balakrishna

Richmond, VA, USA
Email: *bbsbems[at]gmail.com*

Abstract: ***This paper highlights the critical need to strengthen the security of Amazon Web Services (AWS) Elastic Compute Cloud (EC2) instances by strategically migrating third-party applications from Instance Metadata Service version 1 (IMDSv1) to version 2 (IMDSv2). The approach employs proxying, showcasing innovation in the migration process. The significance of this methodology becomes apparent when organizations must address the imperative of upgrading third-party software to utilize IMDSv2 calls on EC2 instances. The paper introduces an algorithmic solution in response to potential cost implications associated with such upgrades. This solution intercepts IMDSv1 calls initiated by third-party applications, gathers metadata, and smoothly transitions to IMDSv2 calls. The result ensures a seamless achievement of the security enhancement through a cost-effective alternative for migration. The complexities of the migration process become especially apparent in environments where several mission-critical applications are intricately dependent on the current infrastructure, making this proxying strategy vital. Recognizing that adopting IMDSv2 is a critical security enhancement, addressing vulnerabilities inherent in IMDSv1, this article presents a comprehensive guide explaining the step-by-step procedure of establishing a proxy. This intermediary enables seamless communication between third-party applications and EC2 instances, speeding the move to IMDSv2. Furthermore, the suggested proxying strategy is helpful when cost considerations and potential disruptions associated with third-party application version upgrades are vital variables. By providing an in-depth examination of the migration process, this paper aims to be a valuable resource, providing practical insights and guidance to organizations looking to strengthen the security posture of their AWS EC2 instances while effectively managing the complexities inherent in such transitions. Importantly, this new concept is scalable and, if necessary, may be deployed globally.***

Keywords: IMDSv1, IMDSv2, EC2, Artifactory, Security

1. Background

1.1 Overview of AWS EC2 and IMDS

Amazon Elastic Computation Cloud (AWS EC2) is a cloud computing web service that provides resizable computation capability. It enables users to run virtual servers, known as instances, on-demand to meet various computing demands. EC2 instances are critical components of scalable and adaptable cloud computing systems, allowing users to host applications, manage workloads, and securely store data.

Within EC2, the Instance Metadata Service (IMDS) is a vital component, providing a RESTful interface that gives critical information about an instance during runtime. IMDS provides dynamic metadata like instance type, IP addresses, and security groups, allowing EC2 applications and services to adapt and optimize to their environment. However, the initial implementation, known as IMDSv1, introduced security vulnerabilities, such as vulnerability to Server-Side Request Forgery (SSRF) attacks.

The necessity for increased security inspired the creation of IMDSv2, which has additional protective features. Only authenticated queries from within the instance can access sensitive metadata, reducing potential vulnerabilities related to IMDSv1. Migrating from IMDSv1 to IMDSv2 is critical for improving the security posture of EC2 instances in the ever-changing cloud computing landscape.

1.2 IMDSv1 Versus IMDSv2[1]

The Amazon Elastic Compute Cloud (AWS EC2) Instance Metadata Service (IMDS) is a vital component that provides dynamic information about the operating instance. This service has two iterations: IMDSv1 (Instance Metadata Service version 1) and IMDSv2 (Instance Metadata Service version 2), with IMDSv2 adding additional security features over its predecessor.

1) IMDSv1

a) *Endpoint*: A fixed URL, http://169.254.169.254/, reaches IMDSv1.
b) *Authentication*: IMDSv1 employs a straightforward, unauthenticated HTTP request protocol. The lack of authentication presents security flaws, rendering it vulnerable to attacks like SSRF (Server-Side Request Forgery).
c) *Metadata Accessibility*: Without authentication or authorization checks, any process running on the instance has access to metadata. This unlimited access raises security concerns, particularly in multi-tenant systems.

2) IMDSv2:

a) *Endpoint*: IMDSv2 can be accessible via a flexible endpoint such as *http://169.254.169.254/* (the default) or a user-defined endpoint. Allows users to set up a custom endpoint for added security.
b) *Authentication*: IMDSv2 adds a robust authentication

system based on IAM (Identity and Access Management) roles[3]. Each request requires a signed token obtained with IAM credentials, giving another layer of protection.

c) *Metadata Accessibility*: Metadata access is restricted by default, needing explicit approval via IAM roles. Adds authentication checks to prevent unwanted access as part of a defense-in-depth approach.

d) *Additional Security Features*: IMDSv2 has a secure token exchange mechanism that ensures only authenticated and authorized processes can access metadata. Supports short-lived session tokens, limiting the window of vulnerability.

3) Migration Considerations:

Backward Compatibility: IMDSv2 builds backward compatibility into its system, enabling a gradual migration approach. Organizations can gradually transition existing programs that use IMDSv1 to IMDSv2 without causing immediate interruption.

4) Security Implications:

The migration from IMDSv1 to IMDSv2 is crucial for addressing security concerns associated with unauthenticated metadata access.

2. Challenges in Upgrading Third-Party Apps

Organizations must upgrade third-party apps to utilize Instance Metadata Service version 2 (IMDSv2) on Amazon Elastic Compute Cloud (AWS EC2) instances. These difficulties include integration issues, potential disruptions, and cost considerations.

a) *Potential Upgrade Licensing Costs:* Upgrading third-party applications may incur additional licensing costs.

b) *Time-Consuming Upgrade Process*: Upgrading third-party applications can be time-consuming.

c) *Complexities in Testing Due to Mission-Critical Dependencies*: Dependence on third-party applications by mission-critical applications complicates the testing process for all applications.

d) *Impact on Planned Activities and Teams Allocation*: The migration process might distract teams from their planned activities.

e) *Ensuring Backward Compatibility*: It is essential to ensure that the upgraded version remains backward compatible with the current tech stack. If not, it will likely have a cascading effect to upgrade the current tech stack before the upgrade.

3. Proposed Solution

To solve the abovementioned concerns, let's write a Python script that provides a solid and safe method for setting up a proxy server to access AWS metadata. The script systematically initializes setups, handles logs, and leverages allowlisting to offer controlled access. Security is prioritized by authenticating user agents, logging critical facts, and properly checking permissions for secure endpoints. The script elegantly includes the IMDSv2 token retrieval mechanism, increasing communication security. A threaded server improves efficiency by allowing concurrent HTTP requests, logging, and response processing. Active signal handling mechanisms ensure a seamless shutdown. The main script expertly manages periodic token fetching, ensuring the proxy server's continuing functionality.

4. Implementation Details

a) Global Variables and Constants

```
# Global Variables and Constants
allowlist = []
secure_allowlist = []
token_cache = ""
token_expiry = datetime.min
token_lock = threading.Lock()  # Renamed from
token_mutex
csv_mutex = threading.Lock()
token_timeout = timedelta(seconds=21600)
token_endpoint =
"http://169.254.169.254/latest/api/token"
config_file_path = "/etc/imds-proxy/"
local_log_file_path = "imds-proxy-output.log"
log_file_path = "/var/log/imds-proxy/imds-proxy.log"
csv_file_path = "/var/log/imds-proxy/useragents.csv"
secure_csv_file_path = "/var/log/imds-
proxy/useragents_security.csv"
other_uas = set()
security_credentials_uas = set()
ttl_header = "X-aws-ec2-metadata-token-ttl-seconds"
blank_ua = "<blank>"
```

Code Snippet 1:
Defines global variables and constants used throughout the script.

b) Initialization

```
# Initialization
def init():
    set_up_logging()
    load_configuration()
```

Code Snippet 2:
The init function is called at the script's entry point to set up logging and load configuration.

c) Logging Configuration

```
# Logging Configuration
def set_up_logging():
    # Set up logging configuration
    pass
```

Code Snippet 3:
The set_up_logging function is a placeholder for setting up the logging configuration.

d) Configuration Loading

```
# Configuration Loading
def load_configuration():
    global allowlist, secure_allowlist
    try:
        with open("config.yaml", "r") as config_file:
            config = yaml.safe_load(config_file)
            allowlist = config.get("allowlist", [])
            secure_allowlist = config.get("secure_allowlist",
[])
    except Exception as e:
        print(f"Error reading config file: {e}")
        sys.exit(1)
```

Code Snippet 4:
The load_configuration function reads the configuration from the config.yaml file and populates the allowlists.

e) CSV Writing

```
# CSV Writing
def write_to_csv(filename, ua):
    with csv_mutex:
        with open(filename, mode="a", newline="",
encoding="utf-8") as file:
            writer = csv.writer(file)
            writer.writerow([ua])
```

Code Snippet 5:
The write_to_csv function writes a user agent (ua) to a CSV file (filename) in a thread-safe manner

f) Secure Endpoint Check

```
# Secure Endpoint Check
def check_secure_endpoint(path):
    secure_endpoints = ["security-credentials"]
    return any(endpoint in path for endpoint in
secure_endpoints)
```

Code Snippet 6:
The check_secure_endpoint function checks if the provided path corresponds to a secure endpoint.

g) User Agent Authentication Check

```
# User Agent Authentication Check
def check_user_agent(ua, is_secure_endpoint):
    is_authenticated = False

    if is_secure_endpoint:
        with csv_mutex:
            if ua not in security_credentials_uas:
                security_credentials_uas.add(ua)
                write_to_csv(secure_csv_file_path, ua)

        if ua in secure_allowlist:
            is_authenticated = True
    else:
        with csv_mutex:
            if ua not in other_uas:
                other_uas.add(ua)
                write_to_csv(csv_file_path, ua)

        for prefix in allowlist:
            if ua.startswith(prefix):
                is_authenticated = True
                break

    return is_authenticated
```

Code Snippet 7:
The check_user_agent function has been updated to handle user agent authentication based on whether it's a secure endpoint or not.
For secure endpoints, it checks if the user agent is in the secure_allowlist and adds it to the security_credentials_uas set.
For non-secure endpoints, it checks if the user agent starts with any prefix in the allowlist and adds it to the other_uas set.
The function returns True if the user agent is authenticated.

h) Request Authorization Check

```
# Request Authorization Check
def check_request_allowed(req):
    ua = req.headers.get("User-Agent", blank_ua)
    path = urlparse(req.path).path

    is_secure_endpoint = check_secure_endpoint(path)
    return check_user_agent(ua, is_secure_endpoint)
```

Code Snippet 8:
The check_request_allowed function checks if the request is allowed based on the user agent and endpoint.

i) IMDSv2 Token Fetching

```
# IMDSv2 Token Fetching
def fetch_imds_v2_token():
    with token_lock:  # Renamed from token_mutex
        global token_cache, token_expiry

        if token_cache and datetime.now() < token_expiry:
            return

        try:
            resp = requests.put(token_endpoint,
headers={ttl_header: "21600"})
            resp.raise_for_status()
            token_cache = resp.text.strip()
            token_expiry = datetime.now() + token_timeout
        except requests.RequestException as e:
            print(f"Failed to fetch IMDSv2 token: {e}")
```

Code Snippet 9:
The fetch_imds_v2_token function fetches the IMDSv2 token if it's not in the cache or has expired.
It uses a lock (token_lock) to ensure thread safety when fetching and updating the token.

j) Proxy Suffix Appending

```
# Proxy Suffix Appending
def append_proxy_suffix(ua):
    return ua + "(imds-proxy)"
```

Code Snippet 10:
The append_proxy_suffix function appends "(imds-proxy)" to a given user agent (ua).

k) Proxy Request Handling

```
# Proxy Request Handling
class ProxyHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        ua = self.headers.get("User-Agent", blank_ua)

        # Log User-agents
        log_fields = {
            "user_agent": ua,
            "method": self.command,
            "uri": self.path,
            "remote_ip": self.client_address[0],
            "headers": dict(self.headers),
        }
        print("Received request", log_fields)

        # Check User-agent is on allow list
        if not check_request_allowed(self):
            print(f"Unauthorized User-Agent detected:
{ua}")
            self.send_response(401)
            self.end_headers()
            return

        # If an IMDS v1 request is made, append the token
for v2 calls
        if self.headers.get("X-aws-ec2-metadata-token") is
None:
            print("Attempting to fetch IMDSv2 token")
            try:
                fetch_imds_v2_token()
                self.headers["X-aws-ec2-metadata-token"] =
token_cache
                print(f"Added token to request:
{token_cache}")
            except Exception as e:
                self.send_response(500)
                self.end_headers()
                self.wfile.write(f"Failed to fetch IMDSv2
token: {e}".encode())
                return

        proxy_request = requests.Request(
            self.command,
f"http://169.254.169.254{self.path}",
headers=dict(self.headers)
        ).prepare()
        response = requests.Session().send(proxy_request)

        self.send_response(response.status_code)
        for key, value in response.headers.items():
            self.send_header(key, value)
        self.end_headers()
        self.wfile.write(response.content)
```

Code Snippet 11:
The ProxyHandler class handles incoming HTTP GET requests.It logs user agent information, checks if the request is allowed, and appends the IMDSv2 token if needed.The response from the IMDS server is then forwarded back to the client.

l) Threaded HTTP Server

```
# Threaded HTTP Server
class ThreadedHTTPServer(ThreadingMixIn,
HTTPServer):
    pass
```

Code Snippet 12:

The ThreadedHTTPServer class is a mix-in class that enables threading for the HTTP server.

m) Proxy Server Execution

```
# Proxy Server Execution
def run_proxy_server():
    server = ThreadedHTTPServer(("127.0.0.1", 9090),
ProxyHandler)
    print("Starting proxy...")
    server.serve_forever()
```

Code Snippet 13:
The run_proxy_server function creates a threaded HTTP server and starts serving requests indefinitely.

n) Signal Handling and Sgutdown

```
# Signal Handling and Shutdown
def shutdown_server(signum, frame):
    print("Shutdown signal received, exiting proxy...")
    sys.exit(0)

if __name__ == "__main__":
    init()

    scheduler = threading.Thread(target=lambda:
time.sleep(5) or fetch_imds_v2_token())
    scheduler.daemon = True
    scheduler.start()

    signal.signal(signal.SIGINT, shutdown_server)
```

```
    signal.signal(signal.SIGTERM, shutdown_server)

    run_proxy_server()
```

Code Snippet 14:
The main block initializes the script, starts a thread to fetch the IMDSv2 token periodically, and sets up signal handlers for graceful shutdown. The proxy server runs in the main thread.

o) Automated Deployment of the above Python Script from JFrog Artifactory on AWS EC2

The EC2 user data section executes the script. It updates the system, installs required packages, sets up JFrog Artifactory, downloads and runs a Python script, and generates user, directory, and iptables rules for secure IMDS proxy operation. The iptables rules specifically divert traffic that does not originate with the designated user ('proxy_user') that is destined for the IMDS IP address ('169.254.169.254') on port 80 to a local destination ('127.0.0.1:9090'). This thorough configuration strengthens the IMDS proxy by ensuring only authorized users can access the metadata service, enhancing overall system integrity.

```
#!/bin/bash
# Function to create directory if it doesn't exist
create_directory() {
    if [ ! -d "$1" ]; then
        mkdir -p "$1"
        chown "$2":"$2" "$1"
        chmod "$3" "$1"
    fi
}
# Function to create user if it doesn't exist
create_user() {
    if id "$1" &>/dev/null; then
        echo "$1 already exists"
    else
        output=$(useradd -r -M -s /sbin/nologin "$1" 2>&1)
        exit_code=$?
    fi
    if [ $exit_code -ne 0 ]; then
        echo "Failed to add user $1. Error: $output"
    else
        echo "User $1 added successfully."
    fi
}
# Update the system and install necessary packages
sudo yum update -y
sudo yum install -y python3 curl iptables
# JFrog Artifactory Configuration
ARTIFACTORY_URL="https://your-artifactory-url.com"
LOCAL_DIRECTORY="/path/to/local/directory"
# Python Script Download and Execution
create_directory "$LOCAL_DIRECTORY" "" 0755
curl -o "$LOCAL_DIRECTORY/imds_proxy.py"
"$ARTIFACTORY_URL/imds_proxy.py"
chmod +x "$LOCAL_DIRECTORY/imds_proxy.py"
pip3 install -r "$LOCAL_DIRECTORY/requirements.txt"
# IMDS Proxy Configuration
LOG_DIR="/var/log/imds-proxy"
CONFIG_DIR="/etc/imds-proxy"
IMDS_PROXY_SERVICE="imds-proxy"
PROXY_USER="proxy_user"
# Install Python (if not already installed)
if ! command -v python3 &>/dev/null; then
    sudo yum install -y python3
fi
# User Creation and iptables Rules
create_user "$PROXY_USER"
yum install -y iptables
/sbin/iptables -t nat -A OUTPUT -m owner ! --uid-owner
"$PROXY_USER" -d 169.254.169.254 -p tcp -m tcp --
dport 80 -j DNAT --to-destination 127.0.0.1:9090
service iptables save
# Log and Config Directory Creation
create_directory "$LOG_DIR" "$PROXY_USER" 0750
create_directory "$CONFIG_DIR" "$PROXY_USER"
0700
# Config.yaml File Creation
cat <<EOL > "$CONFIG_DIR/config.yaml"
allowlist:
    - aws-sdk-
    - Botocore/
    - Boto3/
    - aws-cli/
    - aws-chalice/
    - Java/
secure_allowlist:
    - Java/11.x.xx

EOL
chown "$PROXY_USER":"$PROXY_USER"
"$CONFIG_DIR/config.yaml"
chmod 0600 "$CONFIG_DIR/config.yaml"
# Systemd Service File Creation
cat <<EOL >
"/etc/systemd/system/$IMDS_PROXY_SERVICE.service
"
[Unit]
Description=$IMDS_PROXY_SERVICE Service
After=network.target
[Service]
User=$PROXY_USER
ExecStart=/usr/bin/python3
"$LOCAL_DIRECTORY/imds_proxy.py"
WorkingDirectory=/opt/imds-proxy/
Restart=always
[Install]
WantedBy=multi-user.target
EOL
# Service Startup
systemctl daemon-reload
systemctl enable "$IMDS_PROXY_SERVICE.service"
systemctl start "$IMDS_PROXY_SERVICE.service"
```

Code Snippet 15: Bootstrap Script to run in EC2 User Data

5. Considerations

Use the application's performance test suite to evaluate the proxy server on the EC2 instance (specified in the previous section) on the selected EC2. Throughout the performance test, watch the EC2 CloudWatch metric 'MetadataNoToken'[2]. This statistic is vital since it allows you to evaluate the proxy server's ability to handle metadata requests without tokens. By continuously monitoring this indicator, you can gain insight into the proxy server's performance under varied loads, ensuring it manages metadata requests successfully, even in tokenless settings. This method improves the complete evaluation of the proxy server's capabilities and their impact on the EC2 instance's performance in real-world usage scenarios.

6. Conclusion

This paper illustrates fortifying AWS Elastic Compute Cloud (EC2) instances by transforming third-party applications from IMDSv1 to IMDSv2. The provided Python code snippets outline a user-agent-based solution to challenges such as upgrading third-party software, managing time-consuming migrations, and relying on mission-critical programs. The 'proxyHandler' method actively orchestrates secure transitions, ensuring uninterrupted operations. This technique diminishes security vulnerabilities while concurrently decreasing potential disruptions and licensing costs. The article advocates the global adoption of this strategy, highlighting its strategic significance in enhancing AWS EC2 security. As enterprises confront evolving cloud ecosystems, this white paper acts as a practical guide, delineating a robust strategy to fortify AWS EC2 instances against future risks. Moreover, this approach is adaptable to Java/Javalin or GoLang implementation.

References

[1] AWS (2023, September 28). *Get the full benefits of IMDSv2 and disable IMDSv1 across your AWS infrastructure*. AWS Security Blog. https://aws.amazon.com/blogs/security/get-the-full-benefits-of-imdsv2-and-disable-imdsv1-across-your-aws-infrastructure/
[2] AWS (n.d.). *List the available CloudWatch metrics for your instances*. Amazon Elastic Compute Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html
[3] AWS (n.d.). *Tools for helping with the transition to IMDSv2*. Amazon Elastic Compute Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-metadata-transition-to-version-2.html