GuardDuty Alerting in Slack - Mini Project

0.1 Introduction

This report provides an overview of the development process involved in creating a Lambda function that formats GuardDuty findings and sends them to a designated Slack channel. The Lambda function serves as a bridge between the GuardDuty service and Slack, enabling real-time notification and monitoring of security threats. This report highlights the key steps and challenges encountered during the development process, along with their respective solutions.

0.2 Task 1

Building a Lambda Function: A Comprehensive Explanation

To enhance the accessibility of GuardDuty findings for the Security Team, we implemented a parsing mechanism for SNS messages. These messages are in JSON object format and contain comprehensive information about the findings. The parsed data includes vital details such as the description of the finding, its type, and severity level of the threat.
To ensure the functionality of the code, I initiated the testing process by utilizing the provided webhook. By sending a test message to the designated Slack channel from my local computer, I confirmed that the webhook was operational. The successful delivery of the message indicated that the webhook was functioning correctly. Figure 1.1 shows the test message posted into the slack channel us-west-2.

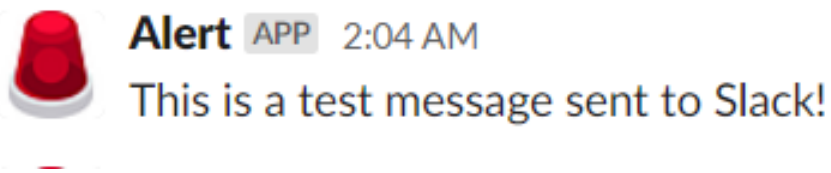

Figure 1: Test Message

To gain a better understanding of the JSON structure of the SNS message, Within the lambda function, I incorporated code to post the event as it is without parsing in the slack channel, then I generated a sample set of findings. These findings would trigger the lambda function to post the event. By generating this sample and implementing the code, I could closely analyze the structure and attributes of the SNS JSON. This approach

ensured that the necessary details could be parsed effectively from the message attribute, enabling the extraction of pertinent information for further processing. Figure 1.2 shows where the provided sample fails to provide information about the SNS notification while Figure 1.3 shows a sample message attribute of the SNS json.

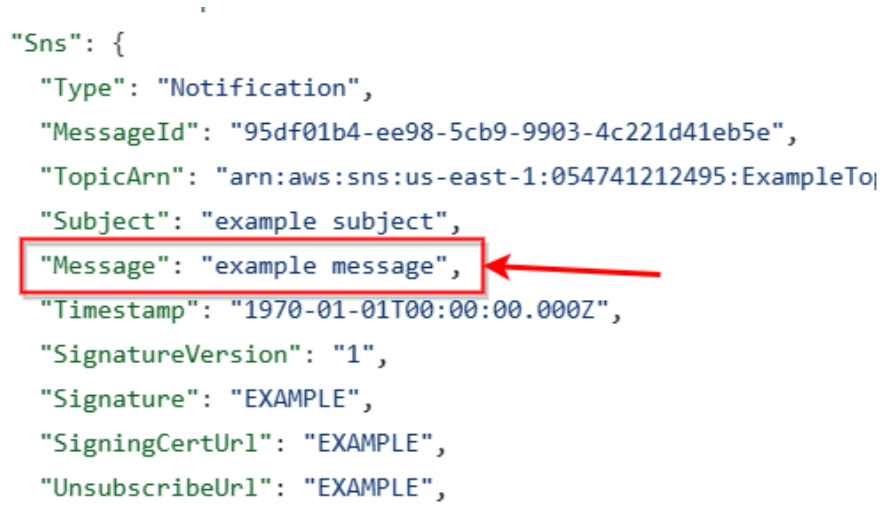

Figure 2: The Provided Sample SNS Notification

"Subject": null,
"Message": "{\"version\":\"0\",\"id\":\"717a27ff-c090-4f61-48f2-671030dd224a\",\"detail-type\":\"GuardDuty Finding\",\"source\":\"aws.guardduty\",\"account\":\"431473634768\",\"time\":\"2023-06-01T15:10:01Z\",\"region\":\"us-west-2\",\"resources\":[],\"detail\":{\"schemaVersion\":\"2.0\",\"accountId\":\"431473634768\",\"region\":\"us-west-2\",\"partition\":\"aws\",\"id\":\"d297af35b8854ab88d35e442ff87e2ac\",\"arn\":\"arn:aws:guardduty:us-west-2:431473634768:detector/8ac43a1ec682771bfdaa743cbf85b0cf/finding/d297af35b8854ab88d35e442ff87e2ac\",\"type\":\"Backdoor:EC2/C&CActivity.B\",\"resource\":{\"resourceType\":\"Instance\",\"instanceDetails\":{\"instanceId\":\"i-99999999\",\"instanceType\":\"m3.xlarge\",\"outpostArn\":\"arn:aws:outposts:us-west-2:123456789000:outpost/op-0fbc006e9abbc73c3\",\"launchTime\":\"2016-08-02T02:05:06.000Z\",\"platform\":null,\"productCodes\":[{\"productCodeId\":\"GeneratedFindingProductCodeId\",\"productCodeType\":\"GeneratedFindingProductCodeType\"}],\"iamInstanceProfile\":

Figure 3: Triggered Sample SNS Notification - Message Attribute

0.2.1 Overview of the Lambda Function Code

The Lambda function parses the GuardDuty finding details, constructs a Slack message with the necessary information, and sends it to the designated Slack channel for the Security Team's review and further action. The code of the Lambda function works as follows:

1. The Lambda handler function, 'lambda_handler', is triggered with an 'event' and 'context' parameter, which contains information about the GuardDuty finding triggered by an SNS message.

2. The code uses JSON parsing to extract relevant information from the 'event' parameter. This includes details such as the AWS account, region, severity, event type, event First Seen, event Last Seen, title, and description. These values are assigned to corresponding variables.

3. Based on the severity level of the finding, the code determines the severity label and color for the Slack message. The severity level is categorized into three levels: LOW, MEDIUM, and HIGH, each associated with a specific color.

4. The code constructs a JSON payload in the required format for Slack. This payload includes various sections and fields that contain the extracted information from the GuardDuty finding, such as the severity, type, AWS account, region, event First Seen, event Last Seen, title, and description.

5. Two buttons are included in the Slack message. The first button allows the recipient to sign in to AWS, and the second button opens the GuardDuty console for further investigation.

6. The Slack webhook URL is fetched from the environment variables. An HTTP POST request is made to the webhook URL using the constructed payload. This sends the message to the specified Slack channel.

7. The response status code is printed to indicate whether the message was successfully sent to Slack or not. This helps in troubleshooting and verifying the status of the Slack message.

Troubleshooting and Challenges

During the process of developing the Lambda function to format GuardDuty findings and send them to the Slack channel in the us-west-2 region, I encountered several errors. Figure 1.4 shows the Monitor of the lambda function:

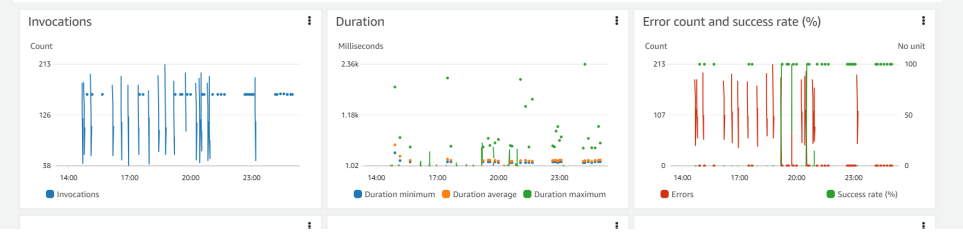

Figure 4: The Monitor of the Lambda Function

These errors were identified and resolved using CloudWatch for monitoring and debugging. Here are some of the some of errors I faced and how I resolved them:

1. **Error with the "requests" library:** Initially, I used the "requests" library to send the parsed information to the Slack channel. However, I encountered an error stating that "requests" has no attribute "post" (Figure 1.5). To resolve this, I switched to using the "urllib3" library, which allowed me to successfully send the formatted information to Slack.

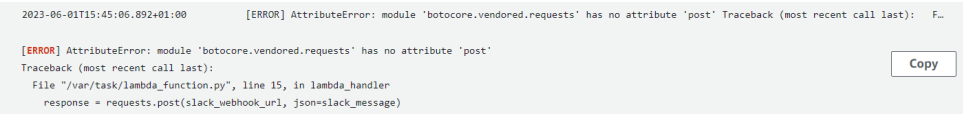

Figure 5: Requests has no Attribute Post Error - Log Feedback

2. **Lambda function not executing:** When checking the logs in CloudWatch, I found that the Lambda function was not executing as expected. Unfortunately, the logs did not provide comprehensive error handling and only returned a status code of 400, indicating a bad request (Figure 1.6).

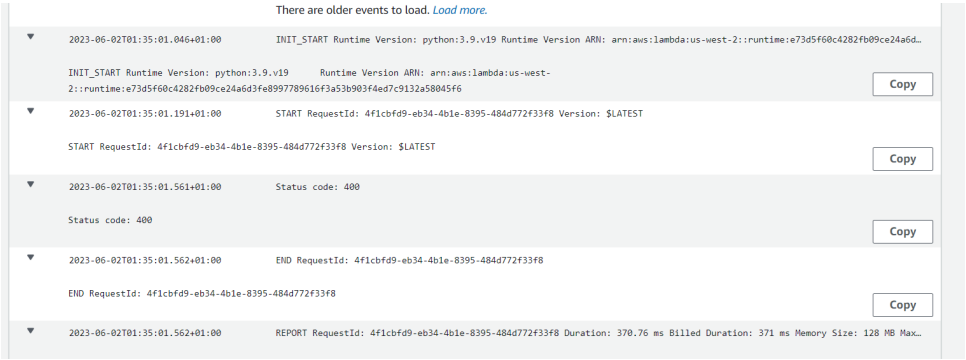

Figure 6: Bad Request Issue - Log Feedback

To identify the source of the error, I focused on the JSON message that was being transmitted to Slack. By systematically deleting each field of the parsed message and triggering the Lambda function, I discovered that a specific JSON field (Figure 1.7) was causing the Slack API to fail processing the POST request with a 400 code. Since this field was not critical, I resolved the error by removing it, allowing Slack to respond to the POST request successfully.

```
            }
        },
        {
            "type": "section",
            "text": {
                "type": "mrkdwn",
                "text": "You can fine-tune alerts by editing the CloudWatch event pattern",
                "verbatim": True,
                "color": "#808080"
            }
        }
    ]
}
]
```

Figure 7: The Source of Bad Request Issue

By actively monitoring and debugging the Lambda function using CloudWatch, I was able to identify and address these errors, ensuring the proper functioning of the function for formatting and transmitting GuardDuty findings to the Slack channel.
Figure 1.8 shows a sample of successfully posted GuardDuty findings in the slack channel us-west-2.

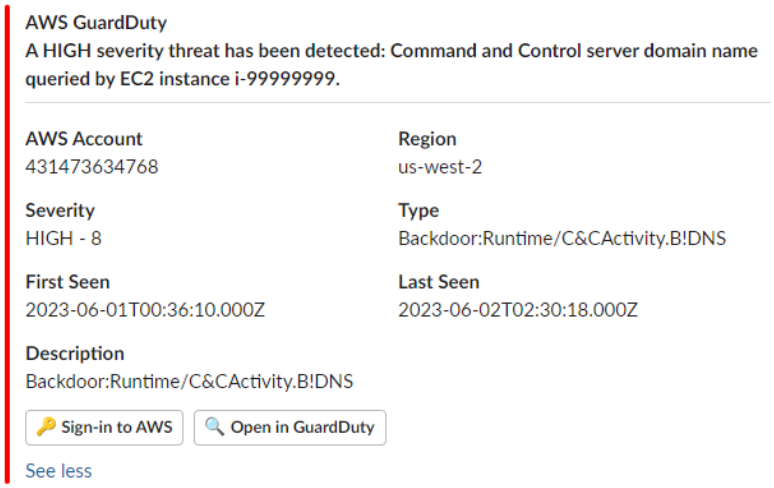

Figure 8: Sample GuardDuty Findings

0.3 Task 2

In this section, we'll explain how to remediate the findings shown in Figure 8. The finding corresponds to a triggered AWS GuardDuty alert of HIGH severity. As per the AWS documentation, this alert indicates a potential compromise of the EC2 instance, with a possibility of unauthorized control by a malicious actor. Such unauthorized control can be exploited for malicious purposes, including the potential for launching DDoS attacks and exploiting the resources of the compromised instance.
To remedy the backdoor threat and mitigate the risk of a compromised EC2 instance, follow these steps:

1. Investigate, identify the specific EC2 instance that has been compromised then isolate it from the network to prevent further unauthorized access.

2. Thoroughly investigate and remove any unauthorized or suspicious files, processes, or configurations associated with the backdoor. The backdoor refers to the malware that allowed unauthorized control over the instance. Then, change credentials and access keys associated with the compromised instance. Reset passwords, rotate access keys, and review/update IAM policies to limit unauthorized access.

3. If unable to identify the backdoor or affected files, I would terminate the compromised instance and create a new one in its place. This ensures an unaffected and secure environment.

0.4 Conclusion

This project involving the development of a Lambda function for formatting GuardDuty findings and integrating with Slack has provided valuable insights into various AWS services and architectures. It enhanced understanding of AWS Lambda, GuardDuty, SNS, JSON parsing, Slack integration, and monitoring with CloudWatch. The project allowed for practical experience in serverless computing, threat detection, data manipulation, real-time communication, and troubleshooting.