2

Vulnerability Assessment

Chatpter description

In this chapter, we will cover the following recipes:

* Creating Vulnerability Assessment Plans
* Threat Assessment using ChatGPT and MITRE ATT&CK Framework
* GPT-assisted vulnerability scanning (API)
* Identifying common vulnerabilities and exposures (CVEs)
* Monitoring emerging vulnerabilities
* Analyzing Vulnerability Assessment Reports
* Prioritizing Vulnerabilities and Remediation Actions
* Using the LangChain framework for large document analysis

Technical Requirements

For this chapter, you will

Creating Vulnerability Assessment Plans

In this recipe, you'll learn how to harness the power of ChatGPT and the OpenAI API to create comprehensive vulnerability assessment plans using network, system, and business details as input. This recipe is invaluable for both cybersecurity students and beginners looking to familiarize themselves with proper methods and tools for vulnerability assessments, as well as experienced cybersecurity professionals aiming to save time on planning and documentation.

Building upon the skills acquired in Chapter 1, you will delve deeper into establishing the system role of a cybersecurity professional specializing in vulnerability assessments. You'll learn how to craft effective prompts that generate well-formatted output using markdown language. This recipe will also expand on the techniques explored in Enhancing Output with Templates (Application: Threat Report) and Formatting Output as a Table (Application: Security Controls Table), enabling you to design prompts that produce the desired output format.

Finally, you'll discover how to use the OpenAI API and Python to generate the vulnerability assessment plan, and then export it as a Microsoft Word file. This recipe will serve as a practical guide for creating detailed and efficient vulnerability assessment plans using ChatGPT and the OpenAI API.

Getting Ready

Before diving into the recipe, it's essential to prepare your environment and gather the necessary resources to ensure a smooth and efficient threat assessment process. Here's a list of steps to help you get ready:

1. Python Library: Install the python-docx library: Ensure you have the python-docx library installed in your Python environment, as it will be used to generate Microsoft Word files. You can install it using the command pip install python-docx.
2. MITRE ATT&CK Framework: Familiarize yourself with the MITRE ATT&CK framework: To make the most of this recipe, it's helpful to have a basic understanding of the MITRE ATT&CK framework. Visit https://attack.mitre.org/ for more information and resources.
3. Sample Threats: Prepare a list of sample threat names, attack campaigns, or adversary groups to use as examples while working through the recipe.

How to do it…

In this section, we will walk you through the process of using ChatGPT to create a comprehensive vulnerability assessment plan tailored to a specific network and organization's needs. By providing the necessary details and using the given system role and prompt, you will be able to generate a well-structured assessment plan.

1. Begin by logging in to your ChatGPT account and navigating to the ChatGPT web UI.
2. Start a new conversation with ChatGPT by clicking the "New chat" button.
3. Enter the following prompt to establish a system role:

You are a cybersecurity professional specializing in vulnerability assessment.

1. Enter the following message text, but replace the placeholders in the “{ }” brackets with the appropriate data of your choice. You can either combine this prompt with the system role or enter it separately.

Using cybersecurity industry standards and best practices, create a complete and detailed assessment plan (not a penetration test) that includes: Introduction, outline of the process/methodology, tools needed, and a very detailed multi-layered outline of the steps. Provide a thorough and descriptive introduction and as much detail and description as possible throughout the plan. The plan should not only assessment of technical vulnerabilities on systems but also policies, procedures, and compliance. It should include the use of scanning tools as well as configuration review, staff interviews, and site walk-around. All recommendations should following industry standard best practices and methods. The plan should be a minimum of 1500 words.

Create the plan so that it is specific for the following details:

Network Size: Large

Number of Nodes: 1000

Type of Devices: Desktops, Laptops, Printers, Routers

Specific systems or devices that need to be excluded from the assessment: None

Operating Systems: Windows 10, MacOS, Linux

Network Topology: Star

Access Controls: Role-based access control

Previous Security Incidents: 3 incidents in the last year

Compliance Requirements: HIPAA

Business Critical Assets: Financial data, Personal health information

Data Classification: Highly confidential

Goals and objectives of the vulnerability assessment: To identify and prioritize potential vulnerabilities in the network and provide recommendations for remediation and risk mitigation.

Timeline for the vulnerability assessment: 4 weeks

Team: 3 cybersecurity professionals, including a vulnerability assessment lead and two security analysts

Expected deliverables of the assessment: A detailed report outlining the results of the vulnerability assessment, including identified vulnerabilities, their criticality, potential impact on the network, and recommendations for remediation and risk mitigation.

Audience: The organization's IT department, senior management, and any external auditors or regulators.

Provide the plan using the following format and markdown language:

#Vulnerability Assessment Plan

##Introduction

Thorough Introduction to the plan including the scope, reasons for doing it, goals and objectives, and summary of the plan

##Process/Methodology

Description and Outline of the process/Methodology

##Tools Required

List of required tools and applications, with their descriptions and reasons needed

##Assessment Steps

Detailed, multi-layered outline of the assessment steps

Hint

If you are performing this in the OpenAI Playground, it is advisable to use Chat mode and enter the role in the System window, and the prompt in the User message window.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2.1 – OpenAI Playground Method

1. Review the generated output from ChatGPT. If the output is satisfactory and meets the requirements, you can proceed to the next step. If not, you can either refine your prompt or re-run the conversation to generate a new output.
2. Once you have obtained the desired output, you can use the generated markdown to create a well-structured vulnerability assessment plan in your preferred text editor or markdown viewer.

Example of the resulting ChatGPT output:

A picture containing text, screenshot, font, letter

Description automatically generated

Figure 2.2 – Example ChatGPT Assessment Plan Output

How it works…

The vulnerability assessment plan recipe leverages the power of natural language processing (NLP) and machine learning algorithms to generate a well-structured vulnerability assessment plan. When you provide a specific system role and prompt, ChatGPT uses these inputs to understand the context and generate a response that is tailored to the given role.

By providing the necessary network, system, and business details in the prompt, you guide ChatGPT to generate responses that adhere to the specified structure and formatting. The model leverages its vast knowledge and language understanding capabilities to generate a comprehensive vulnerability assessment plan that follows the industry-standard methodology.

The markdown language output provides a consistent and easy-to-read format for the plan. It includes all the necessary details specific to the given network, system, and business requirements. By using the generated markdown, you can create a well-structured vulnerability assessment plan that is suitable for use in reports, presentations, or other formal documents.

Overall, the vulnerability assessment plan recipe works by leveraging the power of NLP and machine learning algorithms to generate a comprehensive and well-structured plan that adheres to the industry-standard methodology. By using the provided system role and prompt you can streamline the vulnerability assessment process, save time on planning and documentation, and generate professional-grade assessment plans.

There’s more…

In addition to using ChatGPT to generate a vulnerability assessment plan, you can also use the OpenAI API and Python to automate the process. This approach is particularly useful when you have a large number of network configurations to assess or when you need to generate plans on a recurring basis.

The Python script we will present here reads input data from a text file and uses it to fill in the placeholders in the prompt. The resulting markdown output can then be used to create a well-structured vulnerability assessment plan.

While the process is similar to the ChatGPT version, the use of the OpenAI API provides additional flexibility and control over the generated content. Let's dive into the steps involved in the OpenAI API version of the vulnerability assessment plan recipe.

1. Import necessary libraries and set up the OpenAI API:

import openai

import os

from docx import Document

from tqdm import tqdm

import threading

import time

from datetime import datetime

# Set up the OpenAI API

openai.api\_key = os.getenv("OPENAI\_API\_KEY")

In this section, we import the necessary libraries, such as openai, os, docx, tqdm, threading, time, and datetime. We also set up the OpenAI API by providing the API key.

1. Read user input data from a text file:

def read\_user\_input\_file(file\_path: str) -> dict:

user\_data = {}

with open(file\_path, 'r') as file:

for line in file:

key, value = line.strip().split(':')

user\_data[key.strip()] = value.strip()

return user\_data

user\_data\_file = "assessment\_data.txt"

user\_data = read\_user\_input\_file(user\_data\_file)

Here, we define a function read\_user\_input\_file that reads the user input data from a text file and stores it in a dictionary. We then call this function with the assessment\_data.txt file to obtain the user\_data dictionary.

1. Generate the vulnerability assessment plan using the OpenAI API:

def generate\_report(network\_size,

number\_of\_nodes,

type\_of\_devices,

special\_devices,

operating\_systems,

network\_topology,

access\_controls,

previous\_security\_incidents,

compliance\_requirements,

business\_critical\_assets,

data\_classification,

goals,

timeline,

team,

deliverables,

audience: str) -> str:

# Define the conversation messages

messages = [ ... ]

# Call the OpenAI API

response = openai.ChatCompletion.create( ... )

# Return the generated text

return response['choices'][0]['message']['content'].strip()

In this section, we define the generate\_report function, which takes the user input data and calls the OpenAI API to generate the vulnerability assessment plan. The function returns the generated text.

1. Define the API messages:

# Define the conversation messages

messages = [

{"role": "system", "content": "You are a cybersecurity professional specializing in vulnerability assessment."},

{"role": "user", "content": f'Using cybersecurity industry standards and best practices, create a complete and detailed assessment plan ... Detailed outline of the assessment steps'}

]

# Call the OpenAI API

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=messages,

max\_tokens=2048,

n=1,

stop=None,

temperature=0.7,

)

# Return the generated text

return response['choices'][0]['message']['content'].strip()

In the conversation messages, we define two roles: "system" and "user". The "system" role is used to set the context for the AI model, informing it that it's a cybersecurity professional specializing in vulnerability assessment. The "user" role provides the instructions for the AI, which include generating a detailed vulnerability assessment plan based on industry standards, best practices, and the user-supplied data.

The "system" role helps set the stage for the AI, while the "user" role guides the AI in its content generation. This approach follows a similar pattern to the ChatGPT UI section we discussed earlier, where we provided an initial message to the AI to set the context.

For more information on sending API requests and handling responses, please refer to the "Sending API Requests and Handling Responses" recipe. This recipe provides a deeper understanding of interacting with the OpenAI API, including how to structure requests and process the generated content.

1. Convert the generated markdown text to a Word document:

def markdown\_to\_docx(markdown\_text: str, output\_file: str):

document = Document()

# Iterate through the lines of the markdown text

for line in markdown\_text.split('\n'):

# Add headings and paragraphs based on the markdown formatting

...

# Save the Word document

document.save(output\_file)

This function, markdown\_to\_docx, converts the generated markdown text to a Word document. It iterates through the lines of the markdown text, adding headings and paragraphs based on the markdown formatting, and saves the resulting Word document.

1. Display elapsed time while waiting for the API call:

def display\_elapsed\_time():

start\_time = time.time()

while not api\_call\_completed:

elapsed\_time = time.time() - start\_time

print(f"\rCommunicating with the API - Elapsed time: {elapsed\_time:.2f} seconds", end="")

time.sleep(1)

The display\_elapsed\_time function is used to display the elapsed time while waiting for the API call to complete. It uses a loop to print the elapsed time in seconds.

1. Main script execution:

current\_datetime = datetime.now().strftime('%Y-%m-%d\_%H-%M-%S')

assessment\_name = f"Vuln\_ Assessment\_Plan\_{current\_datetime}"

api\_call\_completed = False

elapsed\_time\_thread = threading.Thread(target=display\_elapsed\_time)

elapsed\_time\_thread.start()

try:

# Generate the report using the OpenAI API

report = generate\_report(

user\_data["Network Size"],

user\_data["Number of Nodes"],

user\_data["Type of Devices"],

user\_data["Specific systems or devices that need to be excluded from the assessment"],

user\_data["Operating Systems"],

user\_data["Network Topology"],

user\_data["Access Controls"],

user\_data["Previous Security Incidents"],

user\_data["Compliance Requirements"],

user\_data["Business Critical Assets"],

user\_data["Data Classification"],

user\_data["Goals and objectives of the vulnerability assessment"],

user\_data["Timeline for the vulnerability assessment"],

user\_data["Team"],

user\_data["Expected deliverables of the assessment"],

user\_data["Audience"]

)

api\_call\_completed = True

elapsed\_time\_thread.join()

except Exception as e:

api\_call\_completed = True

elapsed\_time\_thread.join()

print(f"\nAn error occurred during the API call: {e}")

exit()

# Save the report as a Word document

docx\_output\_file = f"{assessment\_name}\_report.docx"

# Handle exceptions during the report generation

try:

with tqdm(total=1, desc="Generating plan") as pbar:

markdown\_to\_docx(report, docx\_output\_file)

pbar.update(1)

print("\nPlan generated successfully!")

except Exception as e:

print(f"\nAn error occurred during the plan generation: {e}")

In the main part of the script, we start by defining the `assessment\_name` based on the current date and time. We then use threading to display the elapsed time while making the API call. The script calls the `generate\_report` function with the user data as arguments, and upon successful completion, it saves the generated report as a Word document using the `markdown\_to\_docx` function. The progress is displayed using the `tqdm` library. If any errors occur during the API call or report generation, they are displayed to the user.

Hint

You can swap out the chat-3.5-turbo model with the GPT-4 model, if you are a ChatGPT Plus subscriber, for often improved results. In fact, GPT-4 is capable of generating a much longer and more detailed generation and/or document. Just keep in mind that the GPT-4 model is a bit more expensive than the chat-3.5-turbo model.

Here is how the completed script should look:

import openai

import os

from docx import Document

from tqdm import tqdm

import threading

import time

from datetime import datetime

# Set up the OpenAI API

openai.api\_key = os.getenv("OPENAI\_API\_KEY")

current\_datetime = datetime.now().strftime('%Y-%m-%d\_%H-%M-%S')

assessment\_name = f"Vuln\_Assessment\_Plan\_{current\_datetime}"

def read\_user\_input\_file(file\_path: str) -> dict:

user\_data = {}

with open(file\_path, 'r') as file:

for line in file:

key, value = line.strip().split(':')

user\_data[key.strip()] = value.strip()

return user\_data

user\_data\_file = "assessment\_data.txt"

user\_data = read\_user\_input\_file(user\_data\_file)

# Function to generate a report using the OpenAI API

def generate\_report(network\_size,

number\_of\_nodes,

type\_of\_devices,

special\_devices,

operating\_systems,

network\_topology,

access\_controls,

previous\_security\_incidents,

compliance\_requirements,

business\_critical\_assets,

data\_classification,

goals,

timeline,

team,

deliverables,

audience: str) -> str:

# Define the conversation messages

messages = [

{"role": "system", "content": "You are a cybersecurity professional specializing in vulnerability assessment."},

{"role": "user", "content": f'Using cybersecurity industry standards and best practices, create a complete and detailed assessment plan (not a penetration test) that includes: Introduction, outline of the process/methodology, tools needed, and a very detailed multi-layered outline of the steps. Provide a thorough and descriptive introduction and as much detail and description as possible throughout the plan. The plan should not only assessment of technical vulnerabilities on systems but also policies, procedures, and compliance. It should include the use of scanning tools as well as configuration review, staff interviews, and site walk-around. All recommendations should following industry standard best practices and methods. The plan should be a minimum of 1500 words.\n\

Create the plan so that it is specific for the following details:\n\

Network Size: {network\_size}\n\

Number of Nodes: {number\_of\_nodes}\n\

Type of Devices: {type\_of\_devices}\n\

Specific systems or devices that need to be excluded from the assessment: {special\_devices}\n\

Operating Systems: {operating\_systems}\n\

Network Topology: {network\_topology}\n\

Access Controls: {access\_controls}\n\

Previous Security Incidents: {previous\_security\_incidents}\n\

Compliance Requirements: {compliance\_requirements}\n\

Business Critical Assets: {business\_critical\_assets}\n\

Data Classification: {data\_classification}\n\

Goals and objectives of the vulnerability assessment: {goals}\n\

Timeline for the vulnerability assessment: {timeline}\n\

Team: {team}\n\

Expected deliverables of the assessment: {deliverables}\n\

Audience: {audience}\n\

Provide the plan using the following format and observe the markdown language:\n\

#Vulnerability Assessment Plan\n\

##Introduction\n\

Introduction\n\

##Process/Methodology\n\

Outline of the process/Methodology\n\

##Tools Required\n\

List of required tools and applications\n\

##Assessment Steps\n\

Detailed outline of the assessment steps'}

]

# Call the OpenAI API

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=messages,

max\_tokens=2048,

n=1,

stop=None,

temperature=0.7,

)

# Return the generated text

return response['choices'][0]['message']['content'].strip()

# Function to convert markdown text to a Word document

def markdown\_to\_docx(markdown\_text: str, output\_file: str):

document = Document()

# Iterate through the lines of the markdown text

for line in markdown\_text.split('\n'):

# Add headings based on the markdown heading levels

if line.startswith('# '):

document.add\_heading(line[2:], level=1)

elif line.startswith('## '):

document.add\_heading(line[3:], level=2)

elif line.startswith('### '):

document.add\_heading(line[4:], level=3)

elif line.startswith('#### '):

document.add\_heading(line[5:], level=4)

# Add paragraphs for other text

else:

document.add\_paragraph(line)

# Save the Word document

document.save(output\_file)

# Function to display elapsed time while waiting for the API call

def display\_elapsed\_time():

start\_time = time.time()

while not api\_call\_completed:

elapsed\_time = time.time() - start\_time

print(f"\rCommunicating with the API - Elapsed time: {elapsed\_time:.2f} seconds", end="")

time.sleep(1)

api\_call\_completed = False

elapsed\_time\_thread = threading.Thread(target=display\_elapsed\_time)

elapsed\_time\_thread.start()

# Handle exceptions during the API call

try:

# Generate the report using the OpenAI API

report = generate\_document(

user\_data["Network Size"],

user\_data["Number of Nodes"],

user\_data["Type of Devices"],

user\_data["Specific systems or devices that need to be excluded from the assessment"],

user\_data["Operating Systems"],

user\_data["Network Topology"],

user\_data["Access Controls"],

user\_data["Previous Security Incidents"],

user\_data["Compliance Requirements"],

user\_data["Business Critical Assets"],

user\_data["Data Classification"],

user\_data["Goals and objectives of the vulnerability assessment"],

user\_data["Timeline for the vulnerability assessment"],

user\_data["Team"],

user\_data["Expected deliverables of the assessment"],

user\_data["Audience"]

)

api\_call\_completed = True

elapsed\_time\_thread.join()

except Exception as e:

api\_call\_completed = True

elapsed\_time\_thread.join()

print(f"\nAn error occurred during the API call: {e}")

exit()

# Save the report as a Word document

docx\_output\_file = f"{assessment\_name}\_report.docx"

# Handle exceptions during the report generation

try:

with tqdm(total=1, desc="Generating plan") as pbar:

markdown\_to\_docx(report, docx\_output\_file)

pbar.update(1)

print("\nPlan generated successfully!")

except Exception as e:

print(f"\nAn error occurred during the plan generation: {e}")

Threat Assessment using ChatGPT and MITRE ATT&CK Framework

In this recipe, you will learn how to leverage ChatGPT and the OpenAI API to conduct a comprehensive threat assessment by providing a threat, attack, or campaign name. By combining the power of ChatGPT with the MITRE ATT&CK framework, you will be able to generate detailed threat reports, tactics, techniques and procedures (TTP) mappings, and associated Indicators of Compromise (IoCs). This information will enable cybersecurity professionals to analyze attack vectors in their environment and extend their capabilities into threat hunting.

Building upon the skills acquired in Chapter 1, this recipe will guide you through establishing the system role of a cybersecurity analyst and engineering effective prompts that generate well-formatted output, including tables. You will learn how to design prompts to obtain the desired output from ChatGPT using both the ChatGPT web UI and a Python script. Additionally, you will learn how to use the OpenAI API to generate a comprehensive threat report in a Microsoft Word file format.

Getting ready

Before diving into the recipe, it's essential to prepare your environment and gather the necessary resources to ensure a smooth and efficient threat assessment process. Here's a list of steps to help you get ready:

1. Python Library: Install the python-docx library: Ensure you have the python-docx library installed in your Python environment, as it will be used to generate Microsoft Word files. You can install it using the command pip install python-docx.
2. MITRE ATT&CK Framework: Familiarize yourself with the MITRE ATT&CK framework: To make the most of this recipe, it's helpful to have a basic understanding of the MITRE ATT&CK framework. Visit https://attack.mitre.org/ for more information and resources.
3. Sample Threats: Prepare a list of sample threat names, attack campaigns, or adversary groups to use as examples while working through the recipe.

How to do it…

By following these steps, you can successfully utilize ChatGPT to generate a detailed threat report using the MITRE ATT&CK Framework and proper markdown formatting. We will be specifying the name of a threat and applying prompt engineering techniques. ChatGPT will then generate a well formatted report with valuable insights that can assist you in threat analysis, attack vector assessment, and even gather IoCs for threat hunting.

1. Begin by logging in to your ChatGPT account and navigating to the ChatGPT web UI.
2. Start a new conversation with ChatGPT by clicking the "New chat" button.
3. Enter the following prompt to establish a system role:

You are a professional cyber threat analyst and MITRE ATT&CK Framework expert.

1. Replace {threat\_name} in the user prompt below with the threat name of your choice (in our example, we will use WannaCry). You can either combine this prompt with the system role or enter it separately.

Provide a detailed report about {threat\_name}, using the following template (and proper markdown language formatting, headings, bold keywords, tables, etc.):

Threat Name (Heading 1)

Summary (Heading 2)

Short executive summary

Details (Heading 2)

Description and details including history/background, discovery, characteristics and TTPs, known incidents

MITRE ATT&CK TTPs (Heading 2)

Table containing all of the known MITRE ATT&CK TTPs that the {threat\_name} attack uses. Include the following columns: Tactic, Technique ID, Technique Name, Procedure (How WannaCry uses it)

Indicators of Compromise (Heading 2)

Table containing all of the known indicators of compromise. Include the following columns: Type, Value, Description

Hint

Just like the previous recipe, you can perform this in the OpenAI Playground and use Chat mode to enter the role in the System window, and the prompt in the User message window.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2.3 – OpenAI Playground Method

1. After entering the appropriate system role and user prompt, click on "ENTER".
2. ChatGPT will process the prompt and generate a formatted threat report with markdown language formatting, headings, bold keywords, tables, and other elements specified in the prompt.

A close-up of a document

Description automatically generated with medium confidence

Figure 2.4 – ChatGPT Threat Report Narrative Output

A close-up of a document

Description automatically generated with low confidence

Figure 2.5 – ChatGPT Threat Report Table Output

1. Review the generated report to ensure it contains the desired information and formatting. If necessary, adjust your user prompt and resubmit it to improve the output.

How it works…

Just as we did in the recipe Recipe 1.4. Applying ChatGPT Roles (Application: AI CISO), when you assign a role to ChatGPT, you provide a specific context or persona for the model to work with. This helps the model generate responses that are tailored to the given role, resulting in more accurate, relevant, and detailed content. The model will generate content that aligns with the expertise and perspective of the assigned role, offering better insights, opinions, or recommendations.

When we provide a threat name and direct ChatGPT to reference the MITRE ATT&CK framework, we are able to leverage it’s massive dataset, which includes detailed information about threats and the MITRE ATT&CK framework. As a result, it is able to correlate the two and quickly give us the relvant threat information as it pertains to the TTPs identified in the framework.

Note

When using the current version of ChatGPT and the OpenAI API as of the time of this writing, the dataset is only trained up through September 2021. Therefore, it will not have knowledge of any threat data after then. However, we will cover techniques later in this book on how to use the API and Python to feed recent data into the request.

By providing a clear template for the output in your prompt, you guide ChatGPT to generate responses that adhere to the specified structure and formatting. This helps ensure that the generated content is consistent, well-organized, and suitable for use in reports, presentations, or other formal documents. The model will focus on generating content that matches the formatting and structure you've provided while still delivering the information you requested. See the recipes Enhancing Output with Templates (Application: Threat Report) and Formatting Output as a Table (Application: Security Controls Table) for further details.

There’s more…

You can extend the power and flexibility of this recipe by using the OpenAI API with a Python script to generate a threat report, similar to the one created in the ChatGPT web UI. Here’s how you do it:

1. Start by importing the necessary libraries:

import openai

import os

from docx import Document

from tqdm import tqdm

import threading

import time

1. Set up the OpenAI API the same as we did in the recipe Setting the OpenAI API Key as an Environment Variable:

openai.api\_key = os.getenv("OPENAI\_API\_KEY")

1. Create a function to generate a report using the OpenAI API:

def generate\_report(threat\_name: str) -> str:

...

return response['choices'][0]['message']['content'].strip()

This function takes a threat name as input and sends it as part of a prompt to the OpenAI API. It returns the generated text from the API response.

1. Create a function to convert the generated text, which is in Markdown format, to a Microsoft Word document:

def markdown\_to\_docx(markdown\_text: str, output\_file: str):

...

document.save(output\_file)

This function takes the generated text in Markdown format and an output file name. It parses the Markdown text and creates a Word document with the appropriate formatting.

1. Create a function to extract tables from the Markdown text:

def extract\_tables(markdown\_text: str):

...

return tables

This function iterates through the Markdown text and extracts any tables it finds.

1. Create a function to display the elapsed time while waiting for the API call:

def display\_elapsed\_time():

...

This function shows the elapsed time in seconds while waiting for the API call to complete.

1. Get the threat name from user input:

threat\_name = input("Enter the name of a cyber threat: ")

1. Start a separate thread to display the elapsed time while making the API call:

api\_call\_completed = False

elapsed\_time\_thread = threading.Thread(target=display\_elapsed\_time)

elapsed\_time\_thread.start()

1. Make the API call and handle exceptions:

try:

report = generate\_report(threat\_name)

api\_call\_completed = True

elapsed\_time\_thread.join()

except Exception as e:

...

1. Save the generated report as a Word document:

docx\_output\_file = f"{threat\_name}\_report.docx"

1. Generate the report and handle exceptions:

try:

with tqdm(total=1, desc="Generating report and files") as pbar:

markdown\_to\_docx(report, docx\_output\_file)

print("\nReport and tables generated successfully!")

except Exception as e:

...

Here is how the completed script should look:

import openai

import os

from docx import Document

from tqdm import tqdm

import threading

import time

# Set up the OpenAI API

openai.api\_key = os.getenv("OPENAI\_API\_KEY")

# Function to generate a report using the OpenAI API

def generate\_report(threat\_name: str) -> str:

# Define the conversation messages

messages = [

{"role": "system", "content": "You are a professional cyber threat analyst and MITRE ATT&CK Framework expert."},

{"role": "user", "content": f'Provide a detailed report about {threat\_name}, using the following template (and proper markdown language formatting, headings, bold keywords, tables, etc.):\n\n\

Threat Name (Heading 1)\n\n\

Summary (Heading 2)\n\

Short executive summary\n\n\

Details (Heading 2)\n\

Description and details including history/background, discovery, characteristics and TTPs, known incidents\n\n\

MITRE ATT&CK TTPs (Heading 2)\n\

Table containing all of the known MITRE ATT&CK TTPs that the {threat\_name} attack uses. Include the following collumns: Tactic, Technique ID, Technique Name, Procedure (How {threat\_name} uses it)\n\n\

Indicators of Compromise (Heading 2)\n\

Table containing all of the known indicators of compromise. Include the following collumns: Type, Value, Description\n\n\ '}

]

# Call the OpenAI API

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=messages,

max\_tokens=2048,

n=1,

stop=None,

temperature=0.7,

)

# Return the generated text

return response['choices'][0]['message']['content'].strip()

# Function to convert markdown text to a Word document

def markdown\_to\_docx(markdown\_text: str, output\_file: str):

document = Document()

# Variables to keep track of the current table

table = None

in\_table = False

# Iterate through the lines of the markdown text

for line in markdown\_text.split('\n'):

# Add headings based on the markdown heading levels

if line.startswith('# '):

document.add\_heading(line[2:], level=1)

elif line.startswith('## '):

document.add\_heading(line[3:], level=2)

elif line.startswith('### '):

document.add\_heading(line[4:], level=3)

elif line.startswith('#### '):

document.add\_heading(line[5:], level=4)

# Handle tables in the markdown text

elif line.startswith('|'):

row = [cell.strip() for cell in line.split('|')[1:-1]]

if not in\_table:

in\_table = True

table = document.add\_table(rows=1, cols=len(row), style='Table Grid')

for i, cell in enumerate(row):

table.cell(0, i).text = cell

else:

if len(row) != len(table.columns): # If row length doesn't match table, it's a separator

continue

new\_row = table.add\_row()

for i, cell in enumerate(row):

new\_row.cells[i].text = cell

# Add paragraphs for other text

else:

if in\_table:

in\_table = False

table = None

document.add\_paragraph(line)

# Save the Word document

document.save(output\_file)

# Function to extract tables from the markdown text

def extract\_tables(markdown\_text: str):

tables = []

current\_table = []

# Iterate through the lines of the markdown text

for line in markdown\_text.split('\n'):

# Check if the line is part of a table

if line.startswith('|'):

current\_table.append(line)

# If the table ends, save it to the tables list

elif current\_table:

tables.append('\n'.join(current\_table))

current\_table = []

return tables

# Function to display elapsed time while waiting for the API call

def display\_elapsed\_time():

start\_time = time.time()

while not api\_call\_completed:

elapsed\_time = time.time() - start\_time

print(f"\rCommunicating with the API - Elapsed time: {elapsed\_time:.2f} seconds", end="")

time.sleep(1)

# Get user input

threat\_name = input("Enter the name of a cyber threat: ")

api\_call\_completed = False

elapsed\_time\_thread = threading.Thread(target=display\_elapsed\_time)

elapsed\_time\_thread.start()

# Handle exceptions during the API call

try:

# Generate the report using the OpenAI API

report = generate\_report(threat\_name)

api\_call\_completed = True

elapsed\_time\_thread.join()

except Exception as e:

api\_call\_completed = True

elapsed\_time\_thread.join()

print(f"\nAn error occurred during the API call: {e}")

exit()

# Save the report as a Word document

docx\_output\_file = f"{threat\_name}\_report.docx"

# Handle exceptions during the report generation

try:

with tqdm(total=1, desc="Generating report and files") as pbar:

markdown\_to\_docx(report, docx\_output\_file)

print("\nReport and tables generated successfully!")

except Exception as e:

print(f"\nAn error occurred during the report generation: {e}")

After executing this script, you will have a threat report in a Word document format, which is similar to the output generated using the ChatGPT web UI. The script demonstrates how you can adapt the techniques learned in the "How to do it..." section to work with the OpenAI API in a Python script.

Hint

You can swap out the chat-3.5-turbo model with the GPT-4 model, if you are a ChatGPT Plus subscriber, for often improved results. Just keep in mind that the GPT-4 model is a bit more expensive than the chat-3.5-turbo model.

You can also improve accuracy and get a more consistent output by lowing the temperature value.

GPT-assisted vulnerability scanning (API)