Phishing Emails

Project plan:

- Research on the topics related to email protection and email content analysis: (present in the requirement analysis section)

Points to research on :

PHISHING EMAIL

-Pattern & Behaviour Based Detection

-Header Based

-SMTPS & Domain Analysis

-Email Content Analysis & Filtering

-Email Security by Configuring SPF, DKIM, and DMARC

-Certificate Verification & Validation

-URL Analysis & Filtering

-DKIM Analysis

-Restrict File Extensions

-Sender Profile Analysis

-Regex Filtering

-IP, Domain & DNS Blacklisting

- Prepare the rough draft of all the logic and the codes necessary to create the module (all the codes are present in the architecture and the design part)

- Complete the creation of the codes and create a plan to integrate all the codes together to create the tool and also deliver it

Requirement Analysis:

Resources required:

Programming language: Python

Python modules:

- whois(domain age and reputation),
- smtplib(port verification),
- re(regex crafting),
- ssl,
- sockets(reverse dns lookup),
- dns.resolver(MX records viewing),
- email(email parsing and analysis),
- openssl,
- pyopenssl,
- imaplib(Email raw data retrieval),
- ipaddress(SPF verification),
- pandas and sklearn(train AI/ML models),
- dnspython(validation of the SPF and publish DMARC records),
- dkim(to handle dkim signing),
- os,
- poplib(to establish connection with the POP3 server)
- mail-parser(source IP and Domain name extraction)
- requests(Updating Blacklists from the web)

AI/ML model training datasets

Domain and IP Blacklist Resources:

- https://github.com/stamparm/ipsum
- https://github.com/hagezi/dns-blocklists
- https://firebog.net/
- https://oisd.nl/setup

- **Header Based Detection**

Header-based detection focuses on analyzing the technical details found in the email header, which contains metadata about the email transmission. Key elements examined include

1. **Email Source Verification**: Checking the email's "Received" headers to trace its path and verify if it originated from a legitimate source or if it has been spoofed.

2. **SPF, DKIM, and DMARC**: These are email authentication protocols used to validate the sender's identity and prevent spoofing:

   SPF (Sender Policy Framework): Checks if the email originated from an authorized mail server for the sender's domain.

   DKIM (DomainKeys Identified Mail): Verifies that the email content has not been altered in transit and originated from the purported sender.

   DMARC (Domain-based Message Authentication, Reporting & Conformance): Provides policies for handling emails that fail SPF or DKIM checks, reducing the risk of phishing.

3. **Header Anomalies**: Detecting irregularities in email headers such as:
   - Missing or forged headers that are typical of phishing attempts.
   - Inconsistencies in timestamps or routing information that suggest the email's origin is suspicious.

Email Security by configuring SPF, DKIM, and DMARC

DMARC, DKIM, and SPF are three email authentication methods. Together, they help prevent spammers, phishers, and other unauthorised parties from sending emails on behalf of a domain* they do not own.

1. Sender Policy Framework (SPF)

- SPF is an email validation system that verifies the authenticity of the sending domain. By setting up SPF records in HubSpot, you can specify the authorised mail servers that are allowed to send emails on behalf of your domain. This helps prevent unauthorised sources from spoofing your domain and protects your reputation as a sender.
- Implementing SPF in HubSpot involves creating a DNS TXT record that lists the IP addresses or hostnames of the authorised mail servers. When an email is received, the recipient's email server checks the SPF record to ensure that the sending server is authorised to send emails on behalf of the domain. If the SPF check fails, the email may be marked as suspicious or rejected altogether.
- By implementing SPF, you can significantly reduce the chances of your emails being marked as spam or phishing attempts. It provides a layer of trust and authenticity, giving your recipients confidence that the email is indeed coming from a legitimate source.

1. DomainKeys Identified Mail (DKIM)

- DKIM is an email authentication method that uses digital signatures to prove the integrity of the email content. By configuring DKIM in HubSpot, you add a unique cryptographic signature to your outgoing emails. Recipients' email servers can then verify the signature and confirm that the message has not been tampered with during transit.
- When you enable DKIM in HubSpot, the platform automatically generates a pair of cryptographic keys: a private key and a public key. The private key is used to sign the outgoing emails, while the public key is published in the DNS records of your domain. When an email is received, the recipient's email server retrieves the public key and uses it to verify the signature. If the signature is valid, it means that the email has not been modified since it was signed by the private key.
- By implementing DKIM, you add an extra layer of security to your emails. It ensures that the content remains intact and unaltered, giving your recipients confidence that the email they received is exactly as you intended it to be.

2. Domain-based Message Authentication, Reporting, and Conformance (DMARC)

- DMARC builds upon SPF and DKIM to provide an additional layer of email authentication. By implementing DMARC in HubSpot, you can specify how email servers should handle emails that fail SPF or DKIM checks. You can choose to monitor, quarantine, or reject such emails, depending on your organisation's security policies.
- When you enable DMARC in HubSpot, you can define a policy that instructs receiving email servers on how to handle emails that do not pass SPF or DKIM authentication. The policy can be set to "none" for monitoring purposes, "quarantine" to send suspicious emails to the recipient's spam folder, or "reject" to outright reject emails that fail authentication.
- DMARC also provides reporting capabilities, allowing you to receive feedback on the authentication status of your emails. You can receive reports that detail which emails

passed or failed authentication, giving you insights into potential spoofing attempts or configuration issues.
- By implementing DMARC, you have greater control over the handling of emails that fail SPF or DKIM checks. It helps protect your brand's reputation by ensuring that only legitimate emails are delivered to your recipients' inboxes.

Email security is a critical aspect of any organisation's digital presence. By implementing SPF, DKIM, and DMARC, you can enhance the trustworthiness of your emails and protect your recipients from phishing attempts and email spoofing.

Steps to Configure SPF Records in Our Tool

1. **Define the SPF Record**

- **Identify Authorised Mail Servers:** Determine which mail servers are allowed to send emails on behalf of your domain. This includes your own servers and any third-party email services you use.

- **Create the SPF Record:** Format the SPF record as a DNS TXT record.

2. **Programmatically Add SPF Record to DNS**

- **DNS Libraries:** Use DNS libraries to modify DNS records within your application. We can use the **dnspython** library.

- **Validate the SPF Record:** After publishing the SPF record, validate it to ensure it is correctly configured. You can build a validation feature within your tool using libraries to check DNS records:

  We can use **dnspython** for validation.

- **Integrate into our tool:**

    - **User Interface**: Provide an interface within our tool where users can specify the IP addresses and domains to include in the SPF record.
    - **Automation**: Automate the process of adding, updating, and validating SPF records based on user input.
    - **Error Handling**: Implement error handling to manage issues during DNS updates or validation.

Example Workflow in Our Tool

- **User Input:**Users enter the IP addresses and domains that should be included in the SPF record.
- **Generate SPF Record:**The tool generates the SPF record string based on user input.
- **Update DNS:**The tool uses DNS libraries to update the TXT record for the domain.
- **Validate SPF Record:**The tool performs a DNS query to verify that the SPF record is correctly published.
- **Report Status:**Provide feedback to the user about the success or failure of the operation.

Steps to Implement DKIM

1. **Generate DKIM Keys:**

   DKIM uses a pair of cryptographic keys: a private key for signing emails and a public key that recipients use to verify the signature.

- **Generate the Keys:** Use a tool like OpenSSL to generate the keys.
- **Store the Private Key:** Securely store the private key on your mail server. This key will be used to sign outgoing emails.
- **Format the Public Key for DNS:** The public key needs to be formatted and added to your DNS records as a TXT record.

2. **Create the DKIM DNS Record:**

- **DKIM Selector:** Choose a DKIM selector, which is a string that helps identify the DKIM public key. Common selectors are simple strings like "default" or "mail".
- **DNS TXT Record:** Add a TXT record to your DNS with the public key.

3. **Sign Outgoing Emails:**

   Integrate DKIM signing into your email sending process. This involves modifying your email headers to include a DKIM-Signature**.**

- **Signing Emails:** Use a library to handle DKIM signing. We can use **dkim** Library.
- **Email Headers:** The dkim.sign function adds a DKIM-Signature header to your email, which recipients use to verify the email's integrity.

4. **Validate DKIM Configuration:**

   After implementing DKIM, validate it to ensure that emails are correctly signed and that recipients can verify the signatures.

Steps to Implement DMARC

1. **Create the DMARC Record:**

A DMARC record is a DNS TXT record that specifies your DMARC policy. This policy tells receiving mail servers how to handle emails that fail SPF and DKIM checks and where to send reports.

2. **Add the DMARC Record to DNS:**

   Publish the DMARC record as a DNS TXT record. Here's how to do it programmatically and manually:

   We can use the dnspython Library to automate the process.

3. **Handle and Analyze DMARC Reports:**

   DMARC generates reports that help you monitor email authentication and detect potential issues. These reports can be handled and analysed programmatically within our tool.

Summary

SPF (Sender Policy Framework), DKIM (DomainKeys Identified Mail), and DMARC (Domain-based Message Authentication, Reporting & Conformance) are essential email authentication protocols designed to prevent email spoofing and phishing. SPF specifies which mail servers are authorised to send emails on behalf of your domain by creating and publishing a DNS TXT record that lists these servers. DKIM adds a cryptographic signature to outgoing emails, which is validated by recipients using a public key published in your DNS records. DMARC builds on SPF and DKIM by allowing domain owners to set policies on how receiving mail servers should handle emails that fail SPF and DKIM checks, and provides a mechanism for reporting back to the domain owner. Implementing these protocols involves setting up and publishing appropriate DNS records, configuring your mail server to sign outgoing emails (for DKIM), and analysing DMARC reports to refine your email authentication setup. Together, these protocols enhance the security and integrity of email communications.

References:

https://www.cloudflare.com/en-in/learning/email-security/dmarc-dkim-spf/
https://support.google.com/a/answer/33786?hl=en
https://datatracker.ietf.org/doc/html/rfc7208
https://datatracker.ietf.org/doc/html/rfc6376

Restricting File Extensions

Dangerous email attachments:

There is very little reason the following extensions should be in legitimate

.adp, .app, .asp, .bas, .bat, .cer, .chm, .cmd, .cnt, .com, .cpl, .crt, .csh, .der, .exe, .fxp, .gadget, .hlp, .hpj, .hta, .inf, .ins, .isp, .its, .js, .jse, .ksh, .lnk, .mad, .maf, .mag, .mam, .maq, .mar, .mas, .mat, .mau, .mav, .maw, .mda, .mdb, .mde, .mdt, .mdw, .mdz, .msc, .msh, .msh1m, .msh2m, .mshxmlm, .msh1xml, .msh2xml, .msi, .msp,.mst, .ops, .osd, .pcd, .pif, .plg, .prf, .prg, .pst, .reg, .scf, .scr, .sct, .shb, .shs, .ps1, .ps1xml, .ps2, .ps2xml, .psc1, .psc2, .tmp, .url, .vb, .vbe, .vbp, .vbs, .vsmacros, .vsw, .ws, .wsc, .wsf, .wsh, .ade, .cla, .class, .grp, .jar, .mcf, .ocx, .pl, .xbap

Malicious attachments can come in several forms:

- EXE files: Directly executable files that can lead to full device access. These files pose the highest risk but are often easily detected.

- JavaScript, VB Script code files, batch (.BAT) files: These are designed to evade detection and will execute code, often to download a program file to compromise a device.

- HTML and PDF files: These files will direct a device to visit a malicious website or contain JavaScript code. In both cases, the result is the same: your browser visits a malicious site to download a malicious payload.

- Microsoft Office files: These files contain malicious macros that execute when you open the file. This remains one of the riskiest types of attachments to open because the malicious code is hidden from surface-level scanners.

- Zipped files: Hackers sometimes include malicious files inside a zip file to get around email filters and scanners.

Steps for implementation:

The code will automatically check the users email for any new emails

It will the check if any file extensions are present in the email and analyze its extension

If any malicious file extension from the list of extensions provided are present then we can create a alert to prevent the user from downloading the file or upright stop the user from downloading the attachment

Ransomware

Ransomware will encrypt your files until you pay a ransom, usually in cryptocurrency. Sometimes, attackers threaten to expose the locked file unless you pay the fee.

Botnet conscription

A botnet is a network of infected computers that a cybercriminal can control to carry out nefarious activities across a linkage of many computers. These activities include:

- DDoS (Distributed Denial of Service) attacks: The botnet will flood a website with an overwhelming amount of traffic, slowing it or bringing it down completely.
- Spam: Sending spam emails from users' personal email accounts.
- Cryptojacking: Mining cryptocurrency using someone else's computer.

APT/backdoor attacks

Malware can lead to an Advanced Persistent Threat (APT) attack, where an intruder accesses a network and remains undetected for an extended period, monitoring and exfiltrating data.

Credential theft

A common purpose of malware is to steal sensitive information such as login credentials, financial info, or personal data to carry out further attacks.

Prevention of malware attacks:

Several layers of protection exist to prevent email attacks.

1. Email filtering: You can configure advanced email filters that detect spam and prevent it from reaching users.
2. Attachment restrictions: This will block certain types of attachments such as .exe files or anything with scripts in them. Legitimate companies will often transfer these types of files in more secure ways.
3. AI monitoring and natural language processing (NLP): AI and NLP can help detect malicious emails based on their content.
4. User privilege management: Users should be blocked from running any software that hasn't been officially installed on their machines. Additionally, the principle of least privilege should be used to make certain users are granted only the permissions needed to fulfill their roles. This ensures that even if an

account is compromised, the attacker's access is limited which helps reduce potential damage by thwarting malware from expanding into the entire network.

IP, Domain and DNS Blacklisting

IP Blacklisting:

IP blacklisting is a technique used to block emails originating from known malicious IP addresses. When an IP address is identified as a source of phishing emails or other malicious activities, it is added to a blacklist maintained by various organizations and services. Email systems and network security devices use these blacklists to prevent emails from these IP addresses from reaching users.

Deployment Points
- Email gateway
- Firewall
- Intrusion Detection/Prevention System

Resources for IP Blacklist:
- https://github.com/hagezi/dns-blocklists/tree/main/ips

Domain Blacklisting:

Domain blacklisting involves blocking emails from domains that are known to be associated with phishing activities. Just like IP blacklisting, domain blacklisting relies on lists of domains that have been identified as malicious by security services.

Deployment Points:
- Email Server/Email Gateway
- Email Filtering Software
- Security Information and Event Management Systems

Resources for Domain Blacklists:
- https://github.com/hagezi/dns-blocklists/tree/main/domains

DNS Blacklisting:

DNS blacklisting (or DNSBL) is a process where certain IP addresses, domain names, or email servers are marked as malicious or spammy.. DNSBLs are databases that provide information about domain reputations via DNS queries, allowing email systems to identify and block malicious domains in real-time.Here Instead of using a DNSBL service we are going to use a In-house generated Blacklist.

Deployment Points:

- DNS Servers
- Email Gateway
- Web Proxy Servers

Based on our requirements https://github.com/hagezi/dns-blocklists is a great resource and most of the content are updated regularly.

~ SMTPS & Domain Analysis

1. SMTPS Validation

Description: Ensure the email is transmitted over a secure, encrypted channel.

Implementation:

- **Port Verification**: Create a module that inspects the email headers to check if the email is sent via SMTPS on ports 465 or 587.
- **SSL/TLS Certificate Verification**: Develop a system to establish a secure connection to the sending server and retrieve the SSL/TLS certificate. Implement a certificate parsing and validation mechanism to ensure the certificate is legitimate and not expired.

2. Domain Age and Reputation

Description: Assess the age and reputation of the sender's domain.

Implementation:

- **WHOIS Lookups**: Implement a WHOIS client within the tool to retrieve the domain registration date and registrar information. Parse the WHOIS response to extract relevant details.
- **Reputation Assessment**: Maintain a local database of known malicious domains and update it periodically based on threat intelligence reports. Cross-reference the sender's domain against this database.

3. MX (Mail Exchange) Records

Description: Validate the MX records to ensure the domain has legitimate mail servers.

Implementation:

- **DNS Lookups**: Implement a DNS resolver within the tool to perform DNS queries and retrieve the MX records of the sender's domain.
- **MX Record Verification**: Check that the MX records point to legitimate mail servers. This can be done by verifying that the IP addresses of the MX records match known good servers or do not belong to known malicious entities.

4. Reverse DNS Lookup

Description: Perform reverse DNS lookups to match the sending IP address with the domain name.

Implementation:

- **Reverse DNS Queries**: Use the tool's DNS resolver to perform reverse DNS lookups on the sending IP address.
- **Consistency Check**: Ensure consistency between the domain in the email header and the domain returned by the reverse DNS query.

5. TLS (Transport Layer Security) Reporting

Description: Implement TLS reporting to monitor the use of secure transmission protocols.

Implementation:

- **MTA-STS Policy**: Develop functionality to retrieve and parse the MTA-STS policy of the sender's domain from DNS.
- **TLS-RPT Implementation**: Implement a reporting mechanism within the tool that captures and logs TLS-related anomalies during email transmission.

Certificate Verification & Validation (CVV)

The issues related to certificate verification and validation:

1. Expired Certificates: Tools check if certificates are still valid. Expired ones can cause security warnings.

2. Invalid Certificates: Ensuring certificates are properly issued by trusted authorities to prevent phishing attempts.

3. Revoked Certificates: Checking if certificates have been revoked due to security concerns.

4. Mismatched Domain Names (Common Name and Subject Alternative Names): Verifying that certificate domain names match the website being accessed to prevent phishing.

5. Weak Key or Signature Algorithms: Detecting and avoiding certificates that use weak encryption algorithms vulnerable to attacks.

6. Self-Signed Certificates: Identifying certificates not issued by trusted authorities, which browsers may not trust.

7. Certificate Chain Issues: Verifying the entire certificate chain for validity to avoid trust errors.

8. Certificate Transparency: Monitoring compliance to ensure all certificates are publicly logged.

9. Extended Validation (EV) Checks: Ensuring certificates have undergone rigorous validation processes for higher assurance.

10. Mis-issued Certificates: Detecting improperly issued certificates that compromise security.

11. Certificate Transparency (CT) Logs Integrity: Ensuring Certificate Transparency logs are maintained accurately and securely.

12. Key Compromise and Reissuance: Detecting compromised private keys and ensuring secure reissuance.

13. Domain Validation Abuse: Monitoring for fraudulent use of domain validation processes to obtain certificates.

14. Multi-domain and Wildcard Certificates: Managing complexities in validating certificates covering multiple domains or using wildcards.

15. Algorithm Deprecation: Identifying and avoiding the use of deprecated cryptographic algorithms.

16. Automated Certificate Management: Monitoring automated processes to ensure certificates are managed securely.

17. Certificate Lifespan Management: Managing certificate expiration and lifespan to maintain security.

18. TLS Version Compatibility: Ensuring certificates support modern TLS versions and secure cipher suites.

19. Private Key Security: Verifying secure storage of private keys to prevent unauthorised access.

20. Security Compliance: Maintaining compliance with evolving cybersecurity standards and best practices.

<u>To solve the problem of certificate verification and validation, use the following algorithms:</u>

1. RSA or Elliptic Curve Cryptography (ECC): For generating digital signatures and encryption in certificates.

2. SHA-256: For hashing to ensure data integrity.

3. OCSP (Online Certificate Status Protocol) or CRL (Certificate Revocation List): To check the revocation status of certificates.

4. TLS Handshake Protocols: Such as RSA, Diffie-Hellman (DH), or Elliptic Curve Diffie-Hellman (ECDH) for secure key exchange.

5. AES (Advanced Encryption Standard): For encrypting data transmission securely in combination with TLS.

These algorithms and protocols ensure secure and trusted communication over the internet by verifying certificate authenticity, maintaining data integrity, and protecting against unauthorized access.

Types of Cases in Certificate Verification & Validation for Phishing Emails:

1. **Self-Signed Certificates:**
    - Phishing emails often use self-signed certificates which are not trusted by browsers and email clients.
    - **Detection:** Self-signed certificates can be detected as they are not issued by a recognized Certificate Authority (CA).
2. **Expired Certificates:**
    - Phishing emails may use certificates that are no longer valid.
    - **Detection:** Verification of the certificate's expiration date against the current date.
3. **Mismatched Domain Names:**
    - The domain name in the certificate does not match the domain name of the sender's email address.
    - **Detection:** Check the Subject Alternative Name (SAN) field or Common Name (CN) field in the certificate.
4. **Certificates Issued by Untrusted Authorities:**
    - Certificates from untrusted or compromised CAs.
    - **Detection:** Cross-checking the CA against a list of trusted CAs.
5. **Compromised Certificates:**
    - Legitimate certificates that have been stolen or compromised.
    - **Detection:** Checking revocation status through CRLs (Certificate Revocation Lists) or OCSP (Online Certificate Status Protocol).

Algorithms Used for Verification & Validation:

1. **Public Key Infrastructure (PKI):**
    - PKI is the underlying framework that supports digital certificates, including X.509 certificates.
2. **Hashing Algorithms:**
    - **SHA-256** or other secure hashing algorithms to ensure the integrity of the certificate.
3. **Cryptographic Algorithms:**
    - **RSA** or **ECDSA** for the public/private key pairs used in the certificates.
4. **OCSP (Online Certificate Status Protocol):**
    - Real-time certificate status checking to see if a certificate is revoked.

5. **Certificate Transparency Logs:**
    - Checking CT logs to verify the legitimacy of a certificate.

Steps to Solve the Problem:

1. **Initial Certificate Check:**
    - Verify the certificate chain to ensure it traces back to a trusted root CA.
    - Check the certificate for validity period and ensure it is not expired.
2. **Domain Name Verification:**
    - Ensure the domain name in the certificate matches the sender's email domain.
3. **Trustworthiness of Issuing CA:**
    - Verify that the certificate is issued by a trusted CA.
4. **Revocation Status:**
    - Use OCSP or CRLs to check if the certificate has been revoked.
5. **Use of Certificate Pinning:**
    - For critical applications, use certificate pinning to associate a host with a specific certificate or public key to prevent man-in-the-middle attacks.

6. **Email Filtering and Analysis:**
    - Implement email filtering systems that can detect phishing attempts using machine learning algorithms.
    - Analyze email headers, URLs, and content for signs of phishing.
7. **User Education:**
    - Train users to recognize phishing attempts and understand the importance of certificate warnings in their email clients.
8. **Advanced Threat Protection:**
    - Use advanced threat protection systems that employ heuristic and behavioral analysis to detect phishing emails and malicious attachments.

Practical Implementation:

1. **Deploy a Secure Email Gateway (SEG):**
    - An SEG can scan incoming emails for phishing content, check certificates, and block suspicious emails.
2. **Regular Updates:**
    - Ensure all systems and software involved in email handling are regularly updated with the latest security patches.
3. **Incident Response Plan:**
    - Have a plan in place to quickly respond to and mitigate phishing attacks, including steps for isolating affected systems and communicating with potentially affected users.

References -
https://www.ssl.com/guide/protect-yourself-from-phishing/

https://cwe.mitre.org/data/definitions/295.html

https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/01-Testing_for_Weak_SSL_TLS_Ciphers_Insufficient_Transport_Layer_Protection

https://www.researchgate.net/publication/341509518_Security_analysis_of_website_certificate_validation_Report_for_the_Computer_Security_at_the_Politecnico_di_Torino

https://www.nccoe.nist.gov/sites/default/files/legacy-files/tls-serv-cert-mgt-nist-sp1800-16b-final.pdf

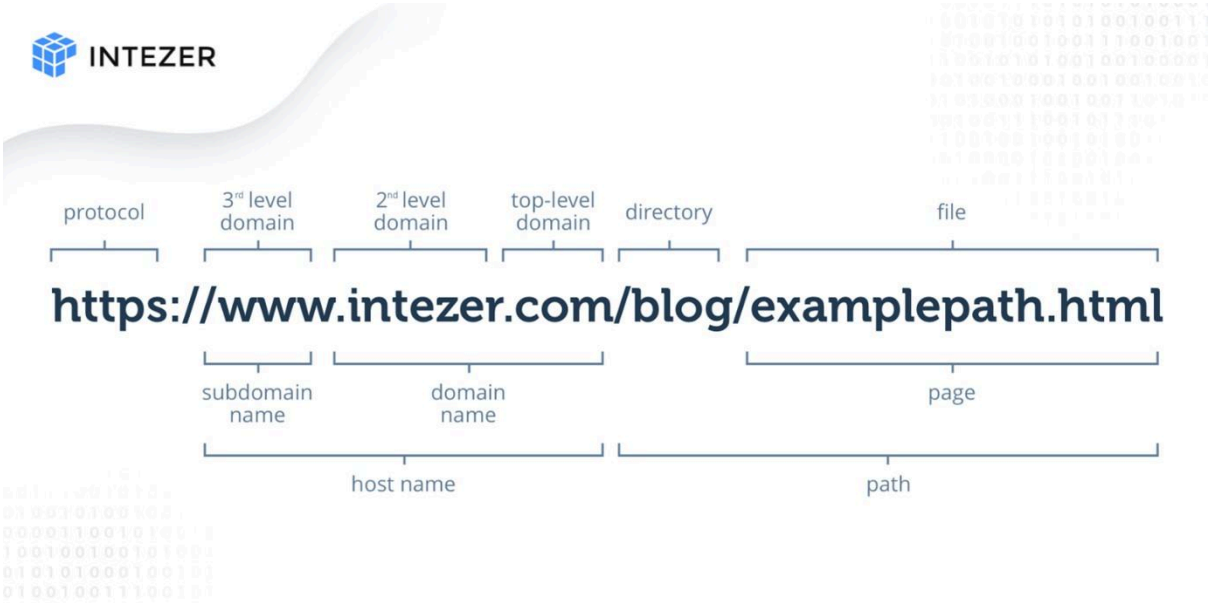
Prevention of phishing emails

Steps to Implement URL Analysis by analyzing Parts of the URL

Gather a Dataset of URLs:

- Collect a dataset of URLs from various sources including publicly available datasets like PhishTank and OpenPhish.
- Include both phishing and legitimate URLs

Develop a tool to break the urls into parts like



Label the URL as phishing or benign by observing factors.

Identify Suspicious Patterns & Identify Common Phishing Patterns:

Analyze phishing URLs to identify common patterns, such as:

URLs containing certain keywords (e.g., "login", "secure", "bank")

URLs using IP addresses instead of domain names

URLs with multiple subdomains or unusual domain structures

URLs with long paths or query strings

URLs containing unusual or excessive special characters

Craft Regex Patterns:

Develop regular expressions (regex) to match the identified patterns.

Examples include:

r'http://\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}' for URLs containing IP addresses

r'https?://[^\s]*login[^\s]*' for URLs containing the word "login"

r'https?://[^\s]*secure[^\s]*' for URLs containing the word "secure"

r'https?://[^\s]*bank[^\s]*' for URLs containing the word "bank"

r'https?://([^\s]+\.){3,}[^\s]+' for URLs with multiple subdomains

r'https?://[^\s]{200,}' for URLs longer than 200 characters

Extract Features:

- Develop a feature extraction process to parse and analyze the URL parts:
  - Protocol (http, https)
  - Domain and subdomains
  - Path

Train a model with Machine Learning Algorithms like SVM,LR based on training data (phishing labelled based on url analysis and crafted regex characters)

Develop the URL Analysis Tool

1. **Parse Incoming Emails:**

- Develop a mechanism to parse incoming emails and extract URLs from email bodies.

2. **Analyze URLs:**

- Implement the feature extraction process on the extracted URLs.
- Use the trained machine learning model to classify each URL as phishing or legitimate.

Use the data we got from the dataset as a training set and use the data we scraped for the user email as the testing set.

For Automation of this tool without manually giving the credentials, we can work on the OAuth2 authentication

OAuth2 Authentication:

- OAuth2 allows applications to access email accounts on behalf of users without requiring the users to share their passwords. Many email providers support OAuth2 for authentication, including Gmail, Outlook, and Yahoo Mail.
- You would need to set up OAuth2 authentication with your email provider and use libraries that support OAuth2 for IMAP, such as `google-auth` for Gmail.

Email Content Analysis and Filtering

Email Body analysis
Grammar and Spelling

Grammatical errors and spelling mistakes are very common in malicious emails coming from non-native speaking individuals, or even ones that are obfuscated intentionally. Examples include incorrect conjugation of verbs and misspelling of words. Genuine emails, especially those coming from professional organizations, would uphold high standards. Automated tools that check for such errors, or even NLP libraries, could be put into action to identify and point out notable ones—elements which will definitely distinguish professional communication from spam.

Word Choice and Frequency

Common trigger words which mostly unfold in most of the spam emails are "urgent," "prize," "free," "money," "winner," and "offer." Analyzing the context and frequency can capture these patterns. Tools of text analysis should be introduced, scanning these keywords and their frequency to identify suspicious emails, comparing them with the norm of regular, legitimate emails.

Contextual Analysis

Relevance and appropriateness of email content is checked against the subject and language used in an email. Any email that does not sound like 'self' or which has totally irreverent content is, therefore, suspicious. Contextual analysis tools will thus check the sender-to-recipient relationship for inconsistencies, which may indicate a spoofing attempt or a compromised account.

Phishing Indicators

Most of the time, phishing emails contain suspicious URLs or requests for personal information. Some of the URLs are very similar to the original, with very minor differences, such as "examp1e.com" instead of "example.com." In most cases, an organization will not request sensitive information over email. The setup of tools that analyze URLs to find minor differences and scan using urgency phrases can thus go a long way to detect these attempts and help protect sensitive information, enhancing cybersecurity

Hex Analysis of Email

Analysis of Suspicious Strings

String Identification:

Scan hexadecimal data for strings that end up being malicious URLs, IP addresses, or even commands hidden in the email content.

Analysis of Encoded Data

Decode Process

Apply decoding algorithms on Base64 or any other encoding techniques applied to attachments to reveal potential malware or links directing users to phishing websites.

Analysis of Double Extensions

Verify Extension

Identify and validate file extensions to identify deceptive attachments—e.g., executable files masked as innocuous document file types.

Concealed Payloads Analysis

Payload Extraction

Extract and analyze the embedded scripts or macros inside email attachments for hidden malware or malicious functionalities.

Obfuscated Code Analysis

Code Decryption

Use hexadecimal analysis to decrypt obfuscated scripts or executable code that reveals underlying malicious activities or payloads.

Property Analysis
Property analysis involves careful scrutiny of the different attributes embedded in an email to determine its authenticity and intention. In analyzing message flags, sender information, recipient information, subject lines, timestamps, Message-ID, and attachments, analysts can identify anomalies that may indicate a security risk. These could include inconsistencies between the 'from' address and the reply-to information, distribution lists larger than usual when using blind carbon copy, urgent subject lines calling for quick action.

Message Flags

Flags are warnings in emails like read and urgent that enable ranking of answers and might identify some that are probably being used in phishing.

Sender Details

Sender's email address, reply-to and return-path fields. This is important in authenticating the sender to avoid spoofing.

Recipient Information

To:, CC:, BCC:, it shows the recipients flagged for the email. This information helps in identifying how wide it was distributed and if perhaps it is some mass mail spam or a phishing campaign.

Subject Line

It contains the subject of the email, which is checked for urgency or other language typical of most phishing emails.

Timestamps

The date and time stamps of when the email was sent are indicated to compare with the sender's usual timings of emailing.

Message ID

A unique identifier for the email; it helps trace the email back to its sender, thread related mails, and also identify emails that are spoofed or their fields manipulated.

Email Hop

The email hop count is simply the number of intermediary servers or MTAs an email message passes through on its way to destination. Every hop an email makes through represents a level in this whole process of delivery, wherein the email message may get forwarded from one server to another based on routing rules and configuration settings. This count, therefore, determines how an email may flow through the internet or organizational networks, outlining the complexity it has to undertake while being delivered from sender to recipient. Hop count is a very key factor in helping email administrators or security staff monitor delivery performance, track routing anomalies, and help spot security threats like unauthorized relaying or e-mail forwarding loops.

Anomalies can be detected
Sudden Increases

A sudden increase in the hop count should be noted; in other words, email routing via unexpected or unauthorized servers.

Routing Inconsistencies

The tool should monitor deviation from normal routing patterns. Examples include emails traversing unusual geographical locations or networks not associated with the sender or recipient.

Proxy Usage

Emails passing through those special proxies that add or modify headers, hence possible modification of the hop count or obfuscation about the origin of an email.

Forwarding Loops

Be on the lookout for emails that are in a stuck-forwarding loop, with them logging continuous cycles through the same servers. This is useful in trying to spot misconfigurations and attempts at email delivery disruption.

Email Header Analysis
From and To fields

The "From" field is the most commonly forged to make the sender's address appear legitimate or as a source that is trusted, and the "To" field, in which emails are diverted to other recipients than the desired ones. Implementation involves validating sender authenticity using e-mail authentication protocols like SPF, DKIM, and DMARC, and scrutinizing recipient addresses for anomalies.

Date and Time

Email timestamps can be manipulated to instill urgency or dupe a recipient. The date and time element will be implemented with server-side time-stamping along with email header time analysis tools, likeantispoofing detection on message time integrity checking.

Received Fields

To hide the route of an e-mail so that it might not be detected, "received" headers may be manipulated. Implementing this checks the email headers for inconsistencies and provides anti-spoofing techniques for authenticity in routing.

X-Originating-IP

This can again be easily modified to hide the email's originating location. Employing the use of IP reputation services and analyzing email headers can identify the actual source IP address and, more importantly, detect attempts that try to masquerade the sender's identity.

MIME-Version and Content-Type

These are also used in disguising dangerous content by using exploits in different MIME versions or content types. This needs to be validated with MIME headers, attachment scanning, and detection through content filtering rules.

References and In-Reply-To field

It can be used to confuse email systems and recipients about the sequence and history of messages in a conversation, making it hard to follow the original thread. Implementation involves analyzing email headers for inconsistencies in thread continuity and verifies the email context to identify phishing attempts or misleading communications.

Attachment Analysis

Verify File Extension

File extension checking is another method to identify probable dangerous attachments. For instance, .exe, .js, .docm, etc. are some risk extensions, which can be found when they arrive as attached files. This can be done by developing a list of hazardous extensions and verifying the attachment against this list, resulting in the flag being automatically raised for further scanning.

Content inspection

It deals with the actual content in a file rather than simply an extension and can reveal hidden threats like embedded scripts or macros. It essentially amounts to parsing and the inspection of file contents by tools for any malicious code or patterns. This is basically deep content inspection libraries implemented on common file types integrated and configured against imparted known threats.

File Metadata Analysis

Metadata forensics reveals details about the source and authenticity of a file, such as information about the author or modification dates. This can be done by extracting metadata using file analysis tools and comparing the information to expected values or known patterns of malicious activity. Implementation will be done by importing libraries to extract metadata and through rule configuration to detect suspicious metadata attributes.

User Alerts and Warnings

Notify users of potentially dangerous attachments to avoid mistakenly running malware files. This is by having warning systems, which upon detection of high-risk attachments, send or trigger warnings to the user to either exercise extreme caution or desist from opening such a file. This mandates integration with the email client or server for raising alerts and advice.

Archived File Analysis

An automated process of extracting and analyzing the content of any archive file sent—such as .zip or .rar —in order to ensure that the malicious files contained are not evaded. This includes the use of libraries to automatically draw out the extracted files recursively for the extraction of the entire archive—with similar methods of inspection applied for the drawn-out extracted files. This involves using libraries that handle various forms of archives while ensuring that all inner files are inspected.

Architecture, Design and Implementation:

All the information about individual components need for the creation of the tool along with the codes are present in this section

`File Extension restriction code POC`

```
import os
```

```
dir_name = “directory path”

test = os.listdir(dir_name)

for item in test:
 if item.endswith(“.exe’):

os.remove(os.path.join(dir_name, item))
```

The code above can be used to delete any file with certain file extension that can be harmful to the clients network. This can work in conjugation with the the file download code below:

```
import poplib

import email

import os

# Replace with your POP3 server details

pop_server = 'your_pop_server'

user = 'your_email@example.com'

password = 'your_password'

# Establish connection to the POP3 server

server = poplib.POP3(pop_server)

server.user(user)

server.pass_(password)

# Get the number of messages in the mailbox

num_messages = len(server.list()[1])

# Loop through each message and retrieve its content

for i in range(num_messages):

# Retrieve the message (index i+1 as POP3 index starts from 1)

resp, lines, octets = server.retr(i+1)

# Convert message lines into a single string

msg_content = b'\n'.join(lines).decode('utf-8', errors='ignore')

# Parse the message using email library

msg = email.message_from_string(msg_content)
```

```
# Iterate over parts of the message (e.g., attachments)

for part in msg.walk():

# Check if part is an attachment

if part.get_content_maintype() == 'multipart':

continue

if part.get('Content-Disposition') is None:

continue

# Retrieve attachment filename

filename = part.get_filename()

if filename:

#add logic to delete the file or open it in a sandbox and analyze it
```

In the code above we can download the email attachments and do whatever we want with the email attachment

---

SMTPS and Domain Analysis

Importing Required Libraries

```
import smtplib  # Used for SMTPS port verification

import ssl  # Used for SSL/TLS certificate verification

import socket  # Used for network operations and reverse DNS lookup

import whois  # Used for retrieving domain registration details

import dns.resolver  # Used for DNS and MX record lookups
```

These libraries provide the functionality needed to perform the various security checks.

Email Security Tool Class

The core of the implementation is encapsulated in the `EmailSecurityTool` class. This class holds methods for each security check and a method to run all checks.

```
class EmailSecurityTool:

    def __init__(self, domain, server, ip):
```

```
        self.domain = domain
        self.server = server
        self.ip = ip

    def run_checks(self):
        self.check_smtps_port()
        self.verify_ssl_certificate()
        self.check_domain_age()
        self.check_domain_reputation()
        self.get_mx_records()
        self.verify_mx_records()
        self.reverse_dns_lookup()
        self.get_mta_sts_policy()
        self.get_tls_rpt_policy()
```

SMTPS ValidationPort Verification

```
def check_smtps_port(self):
    try:
    server = smtplib.SMTP_SSL(self.server, 465)
    print(f"{self.server} is using SMTPS on port 465")
    except:
        try:
          server = smtplib.SMTP(self.server, 587)
          server.starttls()
          print(f"{self.server} is using SMTPS on port 587")
    except:
          print(f"{self.server} is not using SMTPS on ports 465 or
587")
```

SSL/TLS Certificate Verification

```
def verify_ssl_certificate(self):

     context = ssl.create_default_context()

     conn = context.wrap_socket(socket.socket(socket.AF_INET),
server_hostname=self.server)

     conn.connect((self.server, 465))

     cert = conn.getpeercert()

     ssl.match_hostname(cert, self.server)

     print(f"SSL/TLS certificate for {self.server} is valid and
not expired.")
```

Domain Age and Reputation

WHOIS Lookups

```
def check_domain_age(self):

     w = whois.whois(self.domain)

     creation_date = w.creation_date

     print(f"Domain {self.domain} was registered on
{creation_date}")
```

Reputation Assessment

```
def check_domain_reputation(self):

     known_malicious_domains = ['malicious.com', 'phishing.com']

     if self.domain in known_malicious_domains:

     print(f"Domain {self.domain} is known for malicious
activity.")

     else:

     print(f"Domain {self.domain} is not listed as malicious.")
```

MX (Mail Exchange) Records

DNS Lookups

```
def get_mx_records(self):
```

```
     try:
     answers = dns.resolver.resolve(self.domain, 'MX')
     for rdata in answers:
          print(f"MX record for {self.domain}: {rdata.exchange}
with priority {rdata.preference}")
     except dns.resolver.NoAnswer:
     print(f"No MX records found for {self.domain}")
```

MX Record Verification

```
def verify_mx_records(self):
     valid_mx_servers = ['mx1.valid.com', 'mx2.valid.com']
     try:
     answers = dns.resolver.resolve(self.domain, 'MX')
     for rdata in answers:
          mx_record = str(rdata.exchange)
          if mx_record in valid_mx_servers:
               print(f"MX record {mx_record} for {self.domain} is
legitimate.")
          else:
               print(f"MX record {mx_record} for {self.domain} is
suspicious.")
     except dns.resolver.NoAnswer:
     print(f"No MX records found for {self.domain}")
```

Reverse DNS Lookup

```
def reverse_dns_lookup(self):
     try:
     result = socket.gethostbyaddr(self.ip)
     print(f"IP address {self.ip} resolves to {result[0]}")
     except socket.herror:
```

```
     print(f"IP address {self.ip} does not resolve to a valid
domain")
```

TLS Reporting

MTA-STS Policy

```
def get_mta_sts_policy(self):

     try:

     answers = dns.resolver.resolve('_mta-sts.' + self.domain,
'TXT')

     for rdata in answers:

          print(f"MTA-STS policy for {self.domain}:
{rdata.strings}")

     except dns.resolver.NoAnswer:

     print(f"No MTA-STS policy found for {self.domain}")
```

TLS-RPT Implementation

```
def get_tls_rpt_policy(self):

     try:

     answers = dns.resolver.resolve('_smtp._tls.' + self.domain,
'TXT')

     for rdata in answers:

          print(f"TLS-RPT policy for {self.domain}:
{rdata.strings}")

     except dns.resolver.NoAnswer:

     print(f"No TLS-RPT policy found for {self.domain}")



tool = EmailSecurityTool(domain='example.com',
server='smtp.example.com', ip='8.8.8.8')

tool.run_checks()
```

URL Analysis and Regex crafting

Get a dataset to train a machine learning model

Prerequisites

You need to have the scikit-learn, pandas, and email libraries installed. You can install them using pip if you haven't already.
- **pip install scikit-learn pandas**

Script

1. Feature Extraction Function
2. Model Training and Loading
3. Email Parsing and URL Extraction
4. Email Analysis and Flagging
5. Connecting to Email Server and Processing Emails

1. Feature Extraction Function

Code :

```
     import re
from urllib.parse import urlparse, parse_qs

def extract_features(url):
    parsed_url = urlparse(url)

    features = {
        'url_length': len(url),
        'hostname_length': len(parsed_url.hostname) if
parsed_url.hostname else 0,
        'path_length': len(parsed_url.path),
        'query_length': len(parsed_url.query),
        'fragment_length': len(parsed_url.fragment),
        'num_special_chars': len(re.findall(r'[@\-_.]', url)),
        'num_subdomains': parsed_url.hostname.count('.') if
parsed_url.hostname else 0,
        'uses_https': int(parsed_url.scheme == 'https'),
        'has_ip': int(bool(re.search(r'\d+\.\d+\.\d+\.\d+',
parsed_url.hostname))) if parsed_url.hostname else 0,
        'num_query_params': len(parse_qs(parsed_url.query)),
        'contains_suspicious_words':
int(bool(re.search(r'login|verify|bank|secure', url,
re.IGNORECASE)))
    }
    return features
```

Model Training and Loading

For simplicity, we will load a pre-trained model. Make sure to train and save your model as shown previously.

Code :

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
import joblib

# Load the pre-trained model
model = joblib.load('phishing_url_model.pkl')

# Example: Function to train and save the model (run this part
separately as needed)
def train_and_save_model(dataset_path):
    dataset = pd.read_csv(dataset_path)
    dataset['features'] = dataset['url'].apply(extract_features)
    features_df = pd.json_normalize(dataset['features'])
    X_train, X_test, y_train, y_test =
train_test_split(features_df, dataset['label'], test_size=0.2,
random_state=42)

    model = RandomForestClassifier(n_estimators=100,
random_state=42)
    model.fit(X_train, y_train)
```

```
    joblib.dump(model, 'phishing_url_model.pkl')

    accuracy = accuracy_score(y_test, model.predict(X_test))
    print(f'Model accuracy: {accuracy}')

# Uncomment to train and save the model
# train_and_save_model('url_dataset.csv')
```

Email Parsing and URL Extraction

Code :

```
     import email
from email.policy import default

def extract_urls(msg):
    urls = []
    for part in msg.walk():
        if part.get_content_type() == 'text/plain':
            text = part.get_payload()
            urls.extend(re.findall(r'http[s]?://\S+', text))
    return urls
```

. Email Analysis and Flagging

Code :

```
     def analyze_email(raw_email):
    msg = email.message_from_string(raw_email, policy=default)
    urls = extract_urls(msg)

    results = []
    for url in urls:
        features = extract_features(url)
        features_df = pd.json_normalize(features)
        prediction = model.predict(features_df)
        results.append((url, prediction[0]))

    return results

# Example: Analyze a raw email
raw_email = """<raw email content>"""  # Replace with actual raw
email content
analysis_results = analyze_email(raw_email)
print(analysis_results)

# Function to flag or block emails based on analysis
def flag_or_block_email(analysis_results):
    for url, is_phishing in analysis_results:
        if is_phishing:
            print(f"Flagged phishing URL: {url}")
            # Here, you could move the email to a spam folder,
mark it as suspicious, or block it.
        else:
            print(f"Legitimate URL: {url}")
```

```
# Run the flagging/blocking process
flag_or_block_email(analysis_results)
```

5. Connecting to Email Server and Processing Emails

```
# Connect to the email server
def connect_to_email(username, password, server='imap.gmail.com'):
    mail = imaplib.IMAP4_SSL(server)
    mail.login(username, password)
    mail.select('inbox')
    return mail

# Fetch and process emails
def fetch_and_process_emails(mail):
    status, messages = mail.search(None, 'ALL')
    email_ids = messages[0].split()

    for e_id in email_ids:
        status, data = mail.fetch(e_id, '(RFC822)')
        raw_email = data[0][1].decode('utf-8')

        analysis_results = analyze_email(raw_email)
        flag_or_block_email(analysis_results)

# Example usage: Connect to email server and process emails
username = 'your-email@example.com'
password = 'your-password'

mail = connect_to_email(username, password)
fetch_and_process_emails(mail)
```

Notes:

1. **Model Training:** The script assumes you have already trained and saved a model (`phishing_url_model.pkl`). The `train_and_save_model` function is provided to train and save the model if needed.
2. **Raw Email Content:** Replace `"<raw email content>"` with the actual raw email content to analyze and flag/block emails.
3. **Handling Emails:** The script can be expanded to integrate with email servers (e.g., using IMAP or POP3) to fetch and process emails automatically.

This script will help you build a basic URL analysis and filtering tool to detect and flag/block phishing emails based on the URLs found in them. Adjust and expand the script as needed for your specific use case and environment.

Certification Verification and Validation Code

Python Code for certificate verification and email parsing, ensure you have the following prerequisites:

1. Python Requirements:

- Ensure Python is installed on your system.

2. Install Dependencies:

- Use `pip` to install required packages:

- pip install cryptography pyopenssl requests

3. Optional Inputs:

- Prepare optional input files (`cert_file`, `email_file`) or URLs (`url`) as specified in the script.

These steps will prepare your environment to execute the script for certificate validation and email parsing. Adjust inputs as needed for testing purposes.

``````

**Import necessary modules for certificate verification, email parsing, and HTTP requests -**

```
import ssl                    # Provides SSL/TLS functionality

import OpenSSL               # Python bindings for OpenSSL library

from cryptography import x509          # Tools for parsing X.509 certificates

from cryptography.hazmat.backends import default_backend  # Default backend for cryptography module

from cryptography.x509.oid import NameOID    # Object identifiers (OIDs) for X.509 certificates
```
```

```
from datetime import datetime       # Date and time manipulation

from email.parser import BytesParser   # Parse bytes into email
objects

from email.policy import default      # Default parsing policy for
email module

import requests             # HTTP library for making requests to
URLs


# Function to load and parse a certificate from file or data

def load_certificate(cert_data=None, cert_file=None):

    if cert_file:

        with open(cert_file, "rb") as f:

            cert_data = f.read()  # Read certificate data from
file

    if not cert_data:

        raise ValueError("Certificate data or file must be
provided.")

# Ensure certificate data is provided

    cert = x509.load_pem_x509_certificate(cert_data,
default_backend())  # Parse PEM encoded certificate

    return cert


# Function to verify certificate chain using system's trusted CAs

def verify_certificate_chain(cert):

    try:

        trusted_cas = ssl.create_default_context().get_ca_certs()
# Load system's trusted CA certificates

        store = OpenSSL.crypto.X509Store()  # Create X.509
certificate store

        for ca in trusted_cas:

            ca_cert =
OpenSSL.crypto.load_certificate(OpenSSL.crypto.FILETYPE_ASN1, ca)
# Load CA certificate
```

```
            store.add_cert(ca_cert)  # Add CA certificate to store


        store_ctx = OpenSSL.crypto.X509StoreContext(store,
OpenSSL.crypto.X509.from_cryptography(cert))  # Create store
context

        store_ctx.verify_certificate()  # Verify certificate
against trusted CAs

        print("Certificate chain is valid.")  # Print success
message if valid

    except Exception as e:

        print(f"Certificate chain verification failed: {e}")  #
Print error message if verification fails



# Function to check if certificate is expired

def check_expiration(cert):

    not_after = cert.not_valid_after  # Get certificate expiration
date

    if not_after < datetime.utcnow():  # Compare expiration date
with current time

        print("Certificate is expired.")  # Print message if
certificate is expired

    else:

        print("Certificate is valid.")  # Print message if
certificate is still valid



# Function to verify domain name against certificate's common name

def verify_domain(cert, domain_name):

    try:

        common_name =
cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value
# Get common name from certificate

        if domain_name == common_name:

            print("Domain name matches the certificate.")  # Print
message if domain matches certificate
```

```
        else:

            print("Domain name does not match the certificate.")
# Print message if domain does not match certificate

    except Exception as e:

        print(f"Domain verification failed: {e}")  # Print error
message if domain verification fails
```

**# Placeholder function to check certificate revocation status using OCSP**

```
def check_revocation_status(cert):

    try:

        print("Checking revocation status via OCSP...")  #
Placeholder message for OCSP status check

        print("Certificate is not revoked.")  # Placeholder
message assuming certificate is not revoked

    except Exception as e:

        print(f"OCSP check failed: {e}")  # Print error message if
OCSP check fails
```

**# Function to read and parse email from file or data**

```
def parse_email(email_data=None, email_file=None):

    if email_file:

        with open(email_file, 'rb') as f:

            email_data = f.read()  # Read email data from file

    if not email_data:

        raise ValueError("Email data or file must be provided.")
# Ensure email data is provided

    email = BytesParser(policy=default).parsebytes(email_data)  #
Parse email bytes into email object

    return email
```

**# Function to fetch and verify SSL certificate of a URL**

```
def fetch_ssl_certificate(url):

    try:

        response = requests.get(url)  # Send HTTP GET request to
URL

        print(f"HTTP Status Code: {response.status_code}")  #
Print HTTP status code of the response



        cert_data = ssl.get_server_certificate((url, 443))  #
Fetch SSL certificate data from server

        cert =
load_certificate(cert_data=cert_data.encode('utf-8'))  # Load and
parse SSL certificate

        return cert  # Return parsed SSL certificate

    except Exception as e:

        print(f"Failed to fetch SSL certificate: {e}")  # Print
error message if fetching SSL certificate fails

        return None  # Return None if fetching SSL certificate
fails



# Predefined paths and URLs for testing (replace with actual paths
and URLs)

cert_file = "path/to/certificate.pem"  # Replace with actual path
to certificate file

domain_name = "example.com"  # Replace with actual domain name for
certificate verification

email_file = "path/to/email.eml"  # Replace with actual path to
email file for parsing

url = "https://example.com"  # Replace with actual URL for
fetching SSL certificate



# Load and parse certificate if cert_file is provided

if cert_file:

    try:

        cert = load_certificate(cert_file=cert_file)  # Load
certificate from file
```

```
        print("Certificate loaded successfully.")  # Print success message if certificate loaded successfully

    except ValueError as e:

        print(f"Error loading certificate: {e}")  # Print error message if loading certificate fails


    # Verify certificate chain if cert is loaded

    if cert:

        verify_certificate_chain(cert)  # Verify certificate chain against trusted CAs

        check_expiration(cert)  # Check if certificate is expired

        verify_domain(cert, domain_name)  # Verify domain name against certificate

        check_revocation_status(cert)  # Check certificate revocation status (placeholder)


# Parse and analyze email if email_file is provided

if email_file:

    try:

        email = parse_email(email_file=email_file)  # Parse email from file

        print("Email parsed successfully.")  # Print success message if email parsed successfully

    except ValueError as e:

        print(f"Error parsing email: {e}")  # Print error message if parsing email fails


# Fetch and verify SSL certificate from URL if URL is provided

if url:

    url_cert = fetch_ssl_certificate(url)  # Fetch SSL certificate from URL

    if url_cert:
```

```
        verify_certificate_chain(url_cert)  # Verify certificate
chain against trusted CAs

        check_expiration(url_cert)  # Check if certificate is
expired

        verify_domain(url_cert, domain_name)  # Verify domain name
against certificate

        check_revocation_status(url_cert)  # Check certificate
revocation status (placeholder)
```

**# Additional steps: Integrate with ML models, update systems, and handle incident response as per above steps**

**IP, DOMAIN & DNS BLACKLISTING**

IP Blacklisting:

IP blacklisting is a technique used to block emails originating from known malicious IP addresses. When an IP address is identified as a source of phishing emails or other malicious activities, it is added to a blacklist maintained by various organisations and services. Email systems and network security devices use these blacklists to prevent emails from these IP addresses from reaching users.

- https://github.com/stamparm/ipsum
- The above repository has threat intelligence feed based on 30+ lists of suspicious and/or malicious IP addresses.
- The lists are automatically retrieved and parsed every 24 hours.
- The above code can be run every 24 hours to download the updated required blacklist.

Domain Blacklisting:

Domain blacklisting involves blocking emails from domains that are known to be associated with phishing activities. Just like IP blacklisting, domain blacklisting relies on lists of domains that have been identified as malicious by security services.

- https://github.com/hagezi/dns-blocklists
- This blocklist for blocking malware, cryptojacking, scam, spam and phishing, which is updated frequently.
- Here we are going to use Threat Intelligence Medium as it ensures alerts with minimal false positives.

Other Resources:

- https://firebog.net/

- https://oisd.nl/setup/

Email Retrieval from the Server:

```
def fetch_latest_email(server, username, password):

	mail = imaplib.IMAP4_SSL(server)# Connect to the server and login using the
	login creds

	mail.login(username, password)

	mail.select('inbox')



	status, messages = mail.search(None, 'ALL')  # Search for the latest email
in the inbox

	email_ids = messages[0].split()



	latest_email_id = email_ids[-1] #Get the latest email ID



	status, msg_data = mail.fetch(latest_email_id, '(RFC822)') #Fetching Email
Data of the latest email

	for response_part in msg_data:

			if isinstance(response_part, tuple):

			raw_email = response_part[1]

			mail.close()

			mail.logout()

			return raw_email
```

#The above Function fetches latest email from the email server.(Code might be redundant as email data is being fetched before for other operations.

Parsing Email data to Extract Source IP and Domain Name and checking against a Blacklist

```
def parse_and_check(raw_email):

	parsed_mail = mailparser.parse_from_bytes(raw_email) # Parse the raw email
using mailparser



	sender_domain = parsed_mail.from_
```

```
      sender_ip = parsed_mail.get_server_ipaddress() #Extract sender's domain and
IP address




      with open('ip_blacklist.txt', 'r') as f: #checking against IP Blacklist

            for line in f:

                        if sender_ip == line.strip():

                        print(f'Sender IP {sender_ip} found in IP blacklist!')


      with open('domain_blacklist.txt', 'r') as f: #checking against Domain
Blacklist

            for line in f:

                        if sender_domain == line.strip():

                        print(f'Sender Domain {sender_domain} found in domain
blacklist!')
```

#The above Function extracts sender IP and Domain name and then checks it against an Inhouse Blacklist for IP addresses and Domain Names.

# Implementation of SPF Verification:

```
import dns.resolver

import ipaddress



def get_spf_record(domain):

    try:

        answers = dns.resolver.resolve(domain, 'TXT')

        for record in answers:

            for txt_string in record.strings:

                if txt_string.startswith(b'v=spf1'):

                    return txt_string.decode('utf-8')
```

```
    except dns.resolver.NoAnswer:
        return None
    except dns.resolver.NXDOMAIN:
        return None


def parse_spf_record(spf_record):
    terms = spf_record.split()
    mechanisms = [term for term in terms if term != 'v=spf1' and
term != '-all']
    return mechanisms


def check_ip_in_spf(ip_address, mechanisms):
    for mechanism in mechanisms:
        if mechanism.startswith('ip4:'):
            ip_range = mechanism.split(':')[1]
            if ipaddress.ip_address(ip_address) in
ipaddress.ip_network(ip_range):
                return True
        elif mechanism.startswith('include:'):
            included_domain = mechanism.split(':')[1]
            included_spf_record = get_spf_record(included_domain)
            if included_spf_record:
                included_mechanisms =
parse_spf_record(included_spf_record)
                if check_ip_in_spf(ip_address,
included_mechanisms):
                    return True
    return False
```

```
def verify_spf(email_address, sending_ip):
    domain = email_address.split('@')[-1]
    spf_record = get_spf_record(domain)
    if not spf_record:
        print(f"No SPF record found for domain {domain}.")
        return False

    mechanisms = parse_spf_record(spf_record)
    if check_ip_in_spf(sending_ip, mechanisms):
        print(f"IP address {sending_ip} is authorized to send
email for {domain}.")
        return True
    else:
        print(f"IP address {sending_ip} is not authorized to send
email for {domain}.")
        return False

# Example Usage
email_address = "[email protected]"
sending_ip = "192.0.2.1"

result = verify_spf(email_address, sending_ip)
print(f"SPF Verification Result: {result}")
```

## Implementation of Email Hop count And MIME version Analysis

```
import imaplib

import email

from email import policy

from email.parser import BytesParser




def count_hops(email_content):

   msg = BytesParser(policy=policy.default).parsebytes(email_content)

   hop_count = 0



   for header in msg.items():



       if header[0].lower() == 'received':

           hop_count += 1



   return hop_count



def analyze_mime_headers(msg):



   mime_version = msg.get('MIME-Version')

   if mime_version:

       print(f"MIME-Version: {mime_version}")

       if mime_version != '1.0':

           print("Alert: Unusual MIME-Version detected! This email may be
suspicious.")



   content_type = msg.get('Content-Type')
```

```
    if content_type:

        print(f"Content-Type: {content_type}")



def get_gmail_emails():



    raw_email = b"""Received: by 2002:a05:7000:1fa1:b0:596:762d:281d with SMTP
id hr33csp1783068mab;

        Tue, 25 Jun 2024 04:47:44 -0700 (PDT)

Received: from mta-86-140.sparkpostmail.com (mta-86-140.sparkpostmail.com.
[192.174.86.140])

        by mx.google.com with ESMTPS id
d2e1a72fcca58-70677f7881dsi4685954b3a.361.2024.06.25.04.47.43

        for <devsh8681@gmail.com>

        (version=TLS1_2 cipher=ECDHE-ECDSA-AES128-GCM-SHA256 bits=128/128);

        Tue, 25 Jun 2024 04:47:43 -0700 (PDT)

From: Simplilearn <updates@simplilearnmailer.com>

To: "devsh8681@gmail.com" <devsh8681@gmail.com>

Subject:

    =?utf-8?B?RmluYWwgUmVtaW5kZXIgfCDwn5OiIEV0aGljYWwgSGFja2luZyBXZWJpbmFy?=

=?utf-8?Q?_Tomorrow?=

Thread-Topic:

    =?utf-8?B?RmluYWwgUmVtaW5kZXIgfCDwn5OiIEV0aGljYWwgSGFja2luZyBXZWJpbmFy?=

=?utf-8?Q?_Tomorrow?=

Thread-Index: ATZBLjNGBO9J08+wm8WapqQeCHGq5Q==

X-MS-Exchange-MessageSentRepresentingType: 1

Date: Tue, 25 Jun 2024 17:17:43 +0530

Message-ID: <E8.FA.23409.F5EAA766@kn.mta1vrest.cc.prd.sparkpost>

List-Unsubscribe:


<mailto:unsubscribe@unsub.spmta.com?subject=unsubscribe:Lp8Wo6n23WcGo1ahBt2xw
```

```
SLnCaaf0X6eejSgGZ30d6g~|eyAicmNwdF90byI6ICJkZXZzaDg2ODFAZ21haWwuY29tIiwgInRlb
mFudF9pZCI6ICJzcGMiLCAiY3VzdG9tZXJfaWQiOiAiMjkxODU5IiwgInN1YmFjY291bnRfaWQiOi
AiMCIsICJtZXNzYWdlX2lkIjogIjY2NzU1ZmFlN2E2NjgyNGIyZjhlIiB9>,<https://unsubscr
ibe.spmta.com/u/DCwsJixhzLMHhohyjnQWLQ~~/AAR0EwA~/RgRoXTNfPFcDc3BjQgpmdV-uema
CSy-OUhNkZXZzaDg2ODFAZ21haWwuY29tWAQAAAAA>

Content-Language: en-US

X-Hashtags: #Newsletters,#Commercial

X-MS-Has-Attach:

X-MS-Exchange-Organization-SCL: -1

X-MS-TNEF-Correlator:

X-MS-Exchange-Organization-RecordReviewCfmType: 0

received-spf: pass (google.com: domain of

msprvs1=19906gr4oiu_i=bounces-291859@spmailtechnol.com designates

192.174.86.140 as permitted sender) client-ip=192.174.86.140;

Content-Type: multipart/alternative;

   boundary="_000_E8FA23409F5EAA766knmta1vrestccprdsparkpost_"

MIME-Version: 1.0

"""


   msg = email.message_from_bytes(raw_email, policy=policy.default)


   print(msg)


   hop_count = count_hops(raw_email)

   print(f"The email hop count is: {hop_count}")


   unusual_hop_count_threshold = 3


   if hop_count >= unusual_hop_count_threshold:
```

```
        print("Alert: Unusual hop count detected! This email may be
suspicious.")



    analyze_mime_headers(msg)



get_gmail_emails()
```

Resources or Modules used are

imaplib: Classes accessed and manipulated via the IMAP protocol to access mailboxes. It is imported in your code but is not in use considering this provided snippet to fetch emails.

email: This is the Python built-in module for handling email messages. This includes their parsing, creating, and manipulating email messages as well as their components.

email.policy This submodule defines classes that define policies for parsing and handling email messages. In the code, policy.default is used to set the default parsing policy.

email.parser.BytesParser This class is an email parser that parses email messages from byte-like objects; for example, bytes or bytearray. In your code, this will be used in parsing raw email content—the raw_email variable.