Weak Cryptography Testing Report

Introduction

Date: Tuesday, August 5, 2025

This report details the findings from the weak cryptography testing performed on the Damn Vulnerable Web Application (DVWA), focusing on the analysis of hashing algorithms used for sensitive data, particularly user passwords, and other cryptographic weaknesses.

8.1 Testing for Weak SSL/TLS Ciphers, Protocols and Certificates (OTG-CRYPST-001)

Test Objective

The objective of this test is to identify if the web server hosting the application uses weak or outdated SSL/TLS configurations, including vulnerable protocols (e.g., SSLv2, SSLv3, TLS 1.0, TLS 1.1), weak ciphersuites (e.g., RC4, DES, 3DES, export-grade ciphers), or improperly configured certificates (e.g., self-signed, expired, weak key size). Such weaknesses can allow attackers to eavesdrop on encrypted communications, tamper with data, or perform man-in-the-middle attacks.

Target Endpoint

This test targets the web server's SSL/TLS configuration, not typically a specific DVWA application endpoint. The primary target is the HTTPS listener of the web server (e.g., Apache, Nginx) that serves DVWA. If DVWA is configured to run over HTTPS, its URL (e.g., `https://localhost/dvwa`) would be the target for analysis.

Methodology

Testing for weak SSL/TLS configurations is primarily performed using specialized external tools that analyze the server's cryptographic capabilities by initiating TLS handshakes and enumerating supported protocols and ciphers. Python's built-in libraries are generally not

used for this type of comprehensive server-side analysis, but rather for client-side interactions. Tools like `sslyze`, `testssl.sh`, or `nmap` with the `ssl-enum-ciphers` script are commonly used.

Important Tools:

- **`sslyze`:** A Python-based SSL/TLS toolkit that can analyze a server's SSL/TLS configuration.
- **`testssl.sh`:** A free command line tool which checks a server's TLS/SSL ciphers, protocols as well as cryptographic vulnerabilities.
- **`nmap`:** The network scanner, with its `ssl-enum-ciphers` script, can also enumerate supported ciphers.

Note: DVWA itself does not inherently demonstrate SSL/TLS vulnerabilities as it's an application, not a server configuration tool. The vulnerabilities would lie in the underlying web server (e.g., Apache) setup.

Step to Reproduce

1. **Ensure HTTPS is Enabled (Optional for DVWA):** For this test to be relevant, the web server hosting DVWA must be configured to serve content over HTTPS. By default, DVWA often runs on HTTP. If not already configured, you would need to set up SSL/TLS for your Apache/Nginx server.
2. **Run an SSL/TLS Scanner:** Execute one of the recommended tools against the DVWA server's HTTPS port (usually 443).
3. **Analyze Scan Results:** Review the output from the scanner for:
    - **Supported Protocols:** Look for SSLv2, SSLv3, TLS 1.0, TLS 1.1. These are considered insecure and should be disabled. Only TLS 1.2 and TLS 1.3 should be enabled.
    - **Supported Ciphersuites:** Identify weak ciphers (e.g., those using RC4, DES, 3DES, or export-grade ciphers). Look for ciphers with small key sizes or known vulnerabilities.
    - **Certificate Details:** Check for self-signed certificates (in production), expired certificates, certificates with weak signature algorithms (e.g., MD5, SHA1), or certificates with key sizes less than 2048 bits.
    - **Vulnerabilities:** The tools will often report known vulnerabilities like Heartbleed, POODLE, BEAST, CRIME, etc., if the server is susceptible.
4. **Determine Result:** If the server supports outdated protocols, weak ciphers, or has certificate issues, it indicates a cryptographic weakness. A secure configuration

should only allow strong, modern protocols and ciphers, and use properly issued and configured certificates.

Log Evidence

Example 1: `sslyze` output showing weak protocol (Illustrative)

```
$ sslyze --regular localhost:443

* Session Resumption:
    With TLS 1.2:
        - Tickets:
            - Server did not send a
NewSessionTicket.
        - TLS 1.2 Session IDs:
            - Server does not support session IDs.

* TLS 1.0:
    - Supported

* TLS 1.1:
    - Supported

* TLS 1.2:
    - Supported

* TLS 1.3:
    - Not Supported

* Preferred Cipher Suite:
    - Server is using: TLS_RSA_WITH_AES_256_CBC_SHA
(0x35)

* Certificate Information:
    - Certificate is self-signed.
    - Certificate expires in 364 days.
    - Key Size: 2048 bits
```

```
              ... (other details) ...
```

Observation: The server supports TLS 1.0 and TLS 1.1, which are outdated and have known vulnerabilities. It also uses a self-signed certificate, which is not suitable for production environments.

Example 2: `testssl.sh` output showing weak cipher (Illustrative)

```
              $ testssl.sh --fast --warnings off localhost:443

              ...
              Testing vulnerabilities
              ...
              RC4       (CVE-2013-2566, CVE-2015-2808)
VULNERABLE      (NOT ok)
              ...
```

Observation: The server is vulnerable to RC4 attacks, indicating that it supports the RC4 cipher, which is considered insecure.

8.2 Testing for Weak Hashing Algorithms (OTG-CRYPST-002)

Test Objective

The objective of this test is to identify if the application uses weak or outdated hashing algorithms for storing sensitive information, especially user passwords. Weak hashing algorithms (e.g., MD5, SHA1 without salting) are susceptible to collision attacks and rainbow table attacks, allowing attackers to easily reverse hashes and compromise user accounts. Modern applications should use strong, slow, and salted hashing functions like bcrypt, scrypt, Argon2, or PBKDF2.

Target Endpoint

Testing for weak hashing algorithms primarily involves examining how user credentials are stored and processed. Relevant target endpoints and areas in DVWA include:

- **Login Page:** `/login.php` (to observe how passwords are sent)
- **User Management/Registration:** If DVWA had a user registration feature, this would be a key area. For DVWA, this often involves direct database inspection.
- **Database Backend:** Direct access to the DVWA database (e.g., MySQL) to inspect the `users` table where passwords are stored.

Methodology

The methodology involves creating test user accounts, observing how the passwords are transmitted during login (if not over HTTPS), and most importantly, inspecting the database to see how these passwords are stored. If direct database access is not possible, one might infer the hashing algorithm by observing the format of stored hashes (e.g., length, character set) or by attempting to crack known hashes using common algorithms.

Python's built-in `hashlib` library can be used to understand and compare different hashing algorithms. Tools like Burp Suite are essential for intercepting and analyzing HTTP requests during the login process. For cracking hashes, tools like Hashcat or John the Ripper would be used, but the primary goal here is identification, not necessarily cracking.

Important Python Snippet (Illustrative - for understanding hashing, not directly for DVWA interaction):

```
import hashlib

def demonstrate_hashing(password):
    print(f"Original Password: {password}")

    # MD5 (Weak and deprecated for passwords)
    md5_hash = hashlib.md5(password.encode()).hexdigest()
    print(f"MD5 Hash:          {md5_hash} (Length:
{len(md5_hash)})")

    # SHA1 (Weak and deprecated for passwords)
    sha1_hash = hashlib.sha1(password.encode()).hexdigest()
    print(f"SHA1 Hash:         {sha1_hash} (Length:
{len(sha1_hash)})")

    # SHA256 (Better, but still needs salting and stretching
```

```
for passwords)
    sha256_hash = hashlib.sha256(password.encode()).hexdigest()
    print(f"SHA256 Hash:       {sha256_hash} (Length:
{len(sha256_hash)})")

    # Example of a simple salt (not cryptographically secure
for demonstration)
    salt = "random_salt_value".encode()
    salted_sha256 = hashlib.sha256(salt +
password.encode()).hexdigest()
    print(f"Salted SHA256:     {salted_sha256}")

    # bcrypt (Recommended for passwords - requires a library
like 'bcrypt')
    # import bcrypt
    # hashed_bcrypt = bcrypt.hashpw(password.encode(),
bcrypt.gensalt())
    # print(f"Bcrypt Hash:       {hashed_bcrypt.decode()}")

demonstrate_hashing("password123")
demonstrate_hashing("admin")
```

Step to Reproduce

1. **Access DVWA:** Ensure DVWA is running and you can access the login page.
2. **Create a Test User (if possible):** If DVWA allows user registration, create a new user with a known password (e.g., `testuser` / `testpassword`). If not, use existing default credentials (e.g., `admin` / `password`).
3. **Inspect Database:**
    - Access the database used by DVWA (e.g., phpMyAdmin for MySQL).
    - Navigate to the DVWA database (often named `dvwa`).
    - Browse the `users` table.
    - Locate the entry for the test user (or default users).
    - Examine the column where the password is stored (often named `password`).
4. **Analyze Stored Password Format:**
    - **Cleartext:** If the password is stored exactly as entered (e.g., `password` is stored as `password`), it is a critical vulnerability.
    - **MD5/SHA1:** Observe the length and format of the hash. MD5 hashes are 32 hexadecimal characters long. SHA1 hashes are 40 hexadecimal characters

long. If these are found, it indicates a weak hashing algorithm.
    - **Salted Hashes:** Look for a separate `salt` column or a salt concatenated with the hash. Even with salting, MD5/SHA1 are weak.
    - **Strong Hashes (e.g., bcrypt):** Strong hashes typically have a specific format (e.g., `$2y$`, `$2a$`, `$2b$` for bcrypt) and are much longer and more complex.
5. **Determine Result:** If passwords are stored in cleartext, or using weak/deprecated hashing algorithms (MD5, SHA1) without proper salting and stretching, it indicates a critical vulnerability. The application should use strong, modern, and slow hashing functions.

Log Evidence

Example 1: DVWA - Low Security (MD5 Hashing)

Database Table: `dvwa.users`

```
+---------+----------+-------------------------------+
| user_id | user     | password                      |
+---------+----------+-------------------------------+
| 1       | admin    | 5f4dcc3b5aa765d61d8327deb882cf99 |
| 2       | guest    | 084e0343a0486ff05530df6c705c8c16 |
+---------+----------+-------------------------------+
```

Observation: The `password` column contains 32-character hexadecimal strings. This is characteristic of MD5 hashes. For example, `5f4dcc3b5aa765d61d8327deb882cf99` is the MD5 hash of `password`. This is a critical vulnerability as MD5 is a fast, unsalted hashing algorithm easily susceptible to rainbow table attacks.

Example 2: DVWA - Medium Security (SHA1 Hashing)

Database Table: `dvwa.users`

```
+---------+----------+-------------------------------
```

```
-----------+
             | user_id | user     | password
|
             +---------+----------+-----------------------------
-----------+
             | 1       | admin    |
d033e22ae348aeb5660fc2140aec35850c4da997 |
             | 2       | guest    |
8d969eef6ecad3c29a3a629280e686061bce252f |
             +---------+----------+-----------------------------
-----------+
```

Observation: The `password` column contains 40-character hexadecimal strings. This is characteristic of SHA1 hashes. For example, `d033e22ae348aeb5660fc2140aec35850c4da997` is the SHA1 hash of `password`. While slightly better than MD5, SHA1 is also considered cryptographically broken for password hashing and is vulnerable to collision attacks.

8.3 Testing for Weak Encryption Algorithms (OTG-CRYPST-003)

Test Objective

The objective of this test is to identify if the application uses weak or inappropriate encryption algorithms for protecting sensitive data at rest (e.g., in configuration files, databases) or in transit (e.g., within application-level protocols, not covered by SSL/TLS). Weak encryption algorithms (e.g., DES, RC4, ECB mode for block ciphers) can be easily broken, leading to unauthorized disclosure or modification of sensitive information. This test also covers the use of custom or proprietary encryption schemes, which are often weaker than well-vetted, standard algorithms.

Target Endpoint

This test is broader and can apply to various parts of the application where sensitive data is encrypted. In the context of DVWA, this might involve:

- **Configuration Files:** If sensitive data (e.g., API keys, database credentials) were encrypted within configuration files.

- **Database Fields:** If specific sensitive fields in the database (beyond passwords) were encrypted.
- **Application Logic:** Any part of the application's code that performs encryption/decryption of data.
- **Inter-component Communication:** If DVWA communicated with other services using custom encrypted protocols.

Note: DVWA primarily focuses on common web vulnerabilities and does not explicitly feature scenarios for testing weak application-level encryption algorithms. This test would typically require code review or traffic analysis of a more complex application.

Methodology

The methodology for this test involves identifying where sensitive data is encrypted and then analyzing the algorithms and modes used. This often requires a combination of:

- **Code Review:** Examining the application's source code (e.g., PHP files in DVWA) to identify cryptographic functions and their parameters.
- **Traffic Analysis:** Using a proxy (like Burp Suite) to intercept and analyze application traffic for custom encryption schemes or encrypted parameters.
- **File System Analysis:** Inspecting configuration files or data files for encrypted content.
- **Database Inspection:** Checking database fields for encrypted data and attempting to determine the encryption method.

Python's `cryptography` library can be used to implement and understand various encryption algorithms, which can aid in identifying weak ones during analysis. However, direct interaction with DVWA to demonstrate this is limited.

Important Python Library (for analysis/understanding, not direct DVWA interaction):

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.kdf.pbkdf2 import
PBKDF2HMAC
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.ciphers import Cipher,
algorithms, modes
from cryptography.hazmat.backends import default_backend
import os
```

```
# Example: Demonstrating a weak encryption mode (ECB) - DO NOT
USE IN PRODUCTION
def encrypt_ecb(key, plaintext):
    cipher = Cipher(algorithms.AES(key), modes.ECB(),
backend=default_backend())
    encryptor = cipher.encryptor()
    padder = padding.PKCS7(algorithms.AES.block_size).padder()
    padded_data = padder.update(plaintext) + padder.finalize()
    ciphertext = encryptor.update(padded_data) +
encryptor.finalize()
    return ciphertext

# Example: Demonstrating a strong encryption mode (CBC with IV)
def encrypt_cbc(key, iv, plaintext):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())
    encryptor = cipher.encryptor()
    padder = padding.PKCS7(algorithms.AES.block_size).padder()
    padded_data = padder.update(plaintext) + padder.finalize()
    ciphertext = encryptor.update(padded_data) +
encryptor.finalize()
    return ciphertext

# This snippet is for understanding and demonstrating
cryptographic concepts,
# not for directly interacting with DVWA's vulnerabilities.
# DVWA does not typically expose application-level encryption
weaknesses directly.
```

Step to Reproduce

1. **Identify Encrypted Data:** Look for any sensitive data that might be encrypted within the application. This could be in URL parameters, hidden form fields, cookies, database entries, or configuration files.
2. **Analyze Encryption Method (Code Review/Traffic Analysis):**
    - If source code is available (as with DVWA), review relevant PHP files for cryptographic functions (e.g., `mcrypt_*` functions, `openssl_*` functions, or custom implementations).

- Intercept HTTP traffic using a proxy and look for any parameters or data that appear to be encrypted. Attempt to identify the encryption algorithm and mode.
- If data is encrypted in the database, try to determine the algorithm used.

3. **Identify Weaknesses:**
   - **Outdated Algorithms:** Look for algorithms like DES, 3DES (unless used in specific secure configurations), RC4, or Blowfish.
   - **Weak Modes:** Identify block cipher modes like ECB (Electronic Codebook), which is insecure for most applications as it does not hide data patterns. Prefer modes like CBC, CTR, or GCM.
   - **Missing Components:** Check for the absence of Initialization Vectors (IVs) or nonces, or their reuse, which can weaken encryption.
   - **Proprietary Algorithms:** Be wary of custom or "home-grown" encryption algorithms, as they are rarely as secure as well-vetted, standard algorithms.
   - **Hardcoded Keys:** Look for encryption keys hardcoded directly into the source code.
4. **Determine Result:** If the application uses weak encryption algorithms, insecure modes, or has other cryptographic implementation flaws, it indicates a vulnerability. Sensitive data should be protected using strong, modern, and properly implemented cryptographic primitives.

Log Evidence

Example 1: Code Snippet showing use of `mcrypt_encrypt` with MCRYPT_RIJNDAEL_128 and MCRYPT_MODE_ECB (Illustrative)

(Hypothetical PHP code from a vulnerable application)

```
<?php
$key = 'ThisIsASecretKey'; // Hardcoded key - BAD!
$plaintext = 'SensitiveData';

// Using MCRYPT_MODE_ECB - BAD!
$ciphertext = mcrypt_encrypt(MCRYPT_RIJNDAEL_128, $key, $plaintext, MCRYPT_MODE_ECB);
echo base64_encode($ciphertext);
?>
```

Observation: The use of `MCRYPT_MODE_ECB` is a critical flaw as it allows identical plaintext blocks to produce identical ciphertext blocks, revealing patterns in the encrypted data. Additionally, the hardcoded key is a severe security risk. (Note: `mcrypt` is deprecated in modern PHP versions).

Example 2: Traffic analysis showing predictable encrypted parameters (Illustrative)

(Intercepted HTTP request)

```
GET /app/profile?
data=AABBCCDD11223344AABBCCDD11223344 HTTP/1.1
Host: example.com
Cookie: session=...
```

Observation: If the `data` parameter consistently produces the same ciphertext for the same plaintext (e.g., "admin" always encrypts to "AABBCCDD11223344"), it strongly suggests the use of ECB mode or a similar deterministic encryption, which is insecure. Further analysis would be needed to confirm the algorithm.