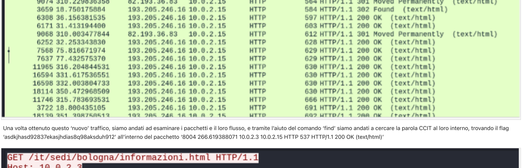
DOCUMENTAZIONE LABORATORIO 4 - FONDAMENTI DI CYBERSECURITY
GRUPPO – LO HACKER

Manuel Castiglia

Thomas Westerman

Esercizio 1

Per risolvere la prima parte, abbiamo inizialmente analizzato il traffico e guardato quali fossero i pacchetti trasmessi, andando a ragionare su quali fossero da escludere e quali fossero da considerare appetibili. Inizialmente abbiamo cercato parole chiavi come CCIT, senza trovare alcun risultato, o parole come FLAG, trovando una ventina di risultati. Ci siamo messi a controllare il flusso http e a fare il 'find' all'interno di ogni pacchetto alla ricerca di CCIT ma senza trovare alcun risultato. Dopo vari tentativi nel cercare la giusta combinazione di filtri per la ricerca, provando a visualizzare tutte le richieste http con il metodo POST o GET (inevitabilmente troppe per fare un controllo singolo su ognuna di loro) o richieste http con esito 200, siamo andati ad applicare questo insieme di filtri: `!http && !ocsp && (!http.content_type contains "text/css" || http.content_type contains "application/javascript") && (!http.content_type contains "image/png" || http.content_type contains "image/jpg") && (http.content_type contains text/html) && (http.response.code == 304)` che va ad eliminare i file css, i file javascript, i file di tipo immagine, i file con codice 304, le richieste di verifica per la validità di un certificato digitale (OCSP) e va a visualizzare i file html e le richieste http.



Una volta ottenuto questo 'nuovo' traffico, siamo andati ad esaminare i pacchetti e il loro flusso, e tramite l'aiuto del comando 'find' siamo andati a cercare la parola CCIT al loro interno, trovando il flag 'asdkjhasd92837ekasjhdiasbq98aksduh912' all'interno dei pacchetti 9004 266.619388071 10.0.2.3 10.0.2.15 HTTP 537 HTTP/1.1 200 OK (text/html)

```
GET /it/sedi/bologna/informazioni.html HTTP/1.1
Host: 10.0.2.3
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1

HTTP/1.1 200 OK
Date: Wed, 01 May 2024 18:08:33 GMT
Server: Apache/2.4.59 (Ubuntu)
Last-Modified: Wed, 01 May 2024 18:08:12 GMT
ETag: "90-617685eabe3a8-gzip"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 147
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

Gentile utente,
ti ringraziamo per aver visualizzato questa pagina.

Il tuo codice .. CCIT{asdkjhasd92837ekasjhdiasbq98aksduh912}

Buon lavoro.
GET /favicon.ico HTTP/1.1
Host: 10.0.2.3
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: image/avif,image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://10.0.2.3/it/sedi/bologna/informazioni.html

HTTP/1.1 200 OK
Date: Wed, 01 May 2024 18:08:33 GMT
Server: Apache/2.4.59 (Ubuntu)
Last-Modified: Wed, 11 May 2022 04:39:14 GMT
ETag: "5760-5deb505f5e080"
Accept-Ranges: bytes
```


Esercizio 2

1. Cross-Site Scripting (XSS)

Codice Vulnerabile (XSS.html):

```
<!DOCTYPE html>
<html>
<head>
    <title>Esempio di XSS</title>
</head>
<body>
    <form method="GET">
        <label for="name">Inserisci il tuo nome:</label>
        <input type="text" id="name" name="name">
        <button type="submit">Invia</button>
    </form>
    <div>
        Ciao, <?php echo $_GET['name']; ?>
    </div>
</body>
</html>
```

Come Sfruttare:

Gli attacchi XSS inietano codice JavaScript malevolo in pagine web viste da altri utenti. Questo codice può rubare cookie, sessioni o eseguire altre azioni dannose per correggere abbiamo inserito `<script>alert('XSS');</script>` nel campo "name" e invia il form. Questo eseguirà l'alert JavaScript quando la pagina viene ricaricata.

Come Correggere:

Escapare correttamente l'output dell'utente usando `htmlspecialchars()` in PHP previene l'esecuzione di script malevoli, htmlspecialchars() infatti converte i caratteri speciali in entità HTML risolvendo quidni la vulnerabilità.

```
<div>
    Ciao, <?php echo htmlspecialchars($_GET['name'], ENT_QUOTES, 'UTF-8'); ?>
</div>
```

2. Cross-Site Request Forgery (CSRF)

Codice Vulnerabile (CSRF.html):

```
<!DOCTYPE html>
<html>
<head>
    <title>Esempio di CSRF</title>
</head>
<body>
    <form method="POST" action="change_name.php">
        <label for="name">Nuovo nome:</label>
        <input type="name" id="name" name="name">
        <button type="submit">Cambia Nome</button>
    </form>
</body>
</html>
```

Come Sfruttare:

Un attaccante induce un utente autenticato a eseguire azioni non desiderate su un sito web in cui è autenticato, in questo caso l'attaccante può creare una pagina malevola che invia automaticamente una richiesta POST a `change_name.php`.

```
<!DOCTYPE html>
<html>
<body>
    <form method="POST" action="http://victima.com/change_name.php">
        <input type="hidden" name="name" value="attacker@example.com">
    </form>
    <script>
        document.forms[0].submit();
    </script>
</body>
</html>
```

Come Correggere:

L'uso di token CSRF assicura che le richieste provengano dall'utente legittimo e non da una fonte esterna. Il token è un valore univoco generato dal server e verificato ad ogni richiesta, per correggere questa vulnerabilità bisogna implementare un token CSRF.

```
// change_email.php
session_start();
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (!hash_equals($_SESSION['token'], $_POST['token'])) {
        die("Validazione del token CSRF fallita");
    }
    // Procedere con il cambio email
}

// CSRF.html (form)
<form method="POST" action="change_email.php">
    <label for="name">Nuovo Nome</label>
    <input type="name" id="name" name="name">
    <input type="hidden" name="token" value="<?php echo $_SESSION['token']; ?>">
    <button type="submit">Cambia Email</button>
</form>
```

3. SQL Injection (SQLi)

Codice Vulnerabile (SQLi.php):

```
<?php
$mysqli = new mysqli("localhost", "utente", "password", "database");
if ($mysqli->connect_error) {
    die('Connessione fallita: ' . $mysqli->connect_error);
}

$id = $_GET['id'];
$result = $mysqli->query("SELECT * FROM utenti WHERE id = '$id'");

while ($row = $result->fetch_assoc()) {
    echo 'Utente: ' . $row['username'];
}
?>
```

Come Sfruttare:

Le iniezioni SQL permettono agli attaccanti di eseguire comandi SQL arbitrari sul database riuscendo a manipolare le query SQL attraverso input non sanificato, in questo caso navigando a `SQLi.php?id=1' OR '1'='1` per recuperare tutti gli utenti.

Come Correggere:

Le dichiarazioni preparate separano i dati dalle query, prevenendo l'iniezione. I placeholder che sono i "(?)" e il binding dei parametri assicurano che gli input vengano trattati come dati, non come parte della query SQL, evitando di eseguire potenziale codice malevolo.

```
$stmt = $mysqli->prepare("SELECT * FROM utenti WHERE id = ?");
$stmt->bind_param("i", $_GET['id']);
$stmt->execute();
$result = $stmt->get_result();

while ($row = $result->fetch_assoc()) {
    echo 'Utente: ' . htmlspecialchars($row['username'], ENT_QUOTES, 'UTF-8');
}
```

4. Command Injection

Codice Vulnerabile (CommandInjection.php):

```
<?php
if (isset($_GET['file'])) {
    $file = $_GET['file'];
    $output = shell_exec('cat ' . $file);
    echo "<pre>$output</pre>";
}
?>

<form method="GET">
    <label for="file">Inserisci il nome del file:</label>
    <input type="text" id="file" name="file">
    <button type="submit">Apri</button>
</form>
```

Come Sfruttare:

Tramite attacchi di iniezione di comandi, gli attaccatti hanno la possibilità' di eseguire codice malevolo sul server. Inserendo, per esempio, `test.txt; ls` nel campo "file" si puo' avere un elenco del contenuto della directory corrente.

Come Correggere:

Validare l'input dell'utente per assicurarsi che contenga solo parametri e caratteri validi e usare la sanificazione aggiungendo escape ai caratteri speciali, `escapeshellarg()` protegge i parametri della shell.

```
if (isset($_GET['file'])) {
    $file = escapeshellarg($_GET['file']);
    $output = shell_exec('cat ' . $file);
    echo "<pre>$output</pre>";
}
```

5. Information Disclosure

Codice Vulnerabile (InfoDisclosure.php):

```
<?php
// Connessione al database
$conn = new mysqli('localhost', "utente", "password", "database");

if ($conn->connect_error) {
    die('Connessione fallita: ' . $conn->connect_error);
}

// Esecuzione di una query
$result = $conn->query("SELECT password FROM utenti WHERE utente = utente");

if (!$result) {
    echo "Errore nella query: " . $conn->error;
} else {
    while ($row = $result->fetch_assoc()) {
        echo 'La password del utente a': " . $row['username'] . "<br>";
    }
}
?>
```

Come Sfruttare:

Se la connessione al database fallisce o se c'è un errore nella query, l'errore dettagliato verrà mostrato all'utente, rivelando informazioni sensibili come il nome del database o le credenziali, venendo quindi esposti ad attacchi.

Come Correggere:

Non mostrare i messaggi di errore dettagliati agli utenti finali. Invece bisogna loggare gli errori in un file sicuro o tramite `error_log()` e mostrare un messaggio generico all'utente.

```
<?php
// Connettiti al database
$conn = new mysqli('localhost', "utente", "password", "database");

if ($conn->connect_error) {
    error_log('Connessione fallita: ' . $conn->connect_error);
    die('Si è verificato un errore.');
}

// Esequire una query
$result = $conn->query("SELECT password FROM utenti WHERE utente = utente");

if (!$result) {
    error_log('Errore nella query: ' . $conn->error);
    echo 'Si è verificato un errore.';
} else {
    while ($row = $result->fetch_assoc()) {
        echo 'La password del utente a': " . htmlspecialchars($row['username'], ENT_QUOTES, 'UTF-8') . "<br>";
    }
}
```