U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

OWASP Top 10 Vulnerabilities

Entrega 2

Turma CC4031_TP2 – Grupo TP2_2

Manuel Ramos Leite Carvalho Neto – up202108744 – up202108744@up.pt

Maria Sousa Carreira – up202408787 – up202408787@up.pt

Matilde Isabel da Silva Simões – up202108782 – up202108782@up.pt

Mestrado em Segurança Informática

Segurança de Redes

2024/2025

Índice

Introdução 2
Arquitetura, Implementação e Tecnologias 2
A01:2021 – Broken Access Control 3
A02:2021 – Cryptographic Failures 8
Conclusão 14

Índice de Figuras

Figura 1 – Página de Perfil do Utilizador “maria” 4
Figura 2 – URL da Página de Perfil com id=1 (“maria”) 4
Figura 3 – Página de Administração 5
Figura 4 – URL da Página de Administração 5
Figura 5 – Contramedida para IDOR 6
Figura 6 – Contramedida para Forced Browsing 7
Figura 7 – URL da Base de Dados 8
Figura 8 – Exploração da Base de Dados com SQLite 9
Figura 9 – Utilização da Ferramenta CrackStation para Quebrar as Hashes 9
Figura 10 – Registo Vulnerável 10
Figura 11 – Autenticação Vulnerável 10
Figura 12 – Pacote de Registo 11
Figura 13 – Credenciais de Registo do Utilizador 11
Figura 14 – Registo Mitigado 12
Figura 15 – Autenticação Mitigada 12
Figura 16 – Ficheiro 000-default.conf 13
Figura 17 – Ficheiro .htaccess 13
Figura 18 – Comunicação por HTTPS 14

Introdução

O OWASP *Top 10* é uma referência globalmente reconhecida para identificar e priorizar os riscos de segurança mais críticos em aplicações *web*. Esta lista, desenvolvida pelo *Open Web Application Security Project* (OWASP), serve como guia para as organizações que procuram reforçar o seu nível de maturidade em cibersegurança. Ao destacar as vulnerabilidades mais frequentes e que causam maior impacto, o OWASP pretende aumentar a consciencialização e promover o desenvolvimento de aplicações *web* mais seguras, utilizando boas práticas e estratégias de mitigação.

Este relatório foca-se nas duas vulnerabilidades já escolhidas anteriormente: **A01:2021 – *Broken Access Control*** e **A02:2021 – *Cryptographic Failures***. Estas vulnerabilidades são bastante críticas devido à sua dominância (ocupando as duas primeiras posições da lista) e ao impacto significativo que podem ter, resultando em consequências graves, como a exposição de informação, a possibilidade de acessos não autorizados ou o comprometimento da integridade dos sistemas.

O objetivo desta segunda parte do trabalho é desenvolver uma aplicação *web* vulnerável e outra mitigada como prova de conceito para demonstrar ataques relacionados com estas vulnerabilidades dentro dos tópicos escolhidos, implementar contramedidas e avaliar a eficácia das mitigações efetuadas.

Arquitetura, Implementação e Tecnologias

O protótipo desenvolvido para a prova de conceito consiste em dois *containers* Docker, em que cada um corre um servidor *web* que aloja um *site*. Existem, portanto, dois *sites*: um vulnerável e outro mitigado. Ambos os *sites* têm as mesmas funcionalidades disponíveis para o utilizador, mas através de implementações e configurações ligeiramente diferentes. Deste modo, uma das versões – a vulnerável – tem falhas que podem ser exploradas por um atacante, mas a outra versão – a mitigada – tem essas falhas corrigidas, resolvendo, assim, as vulnerabilidades em estudo.

Cada *container* corre um servidor Apache, devidamente configurado de acordo com o que se pretende demonstrar. Por um lado, o servidor vulnerável apenas aceita conexões HTTP, encaminhando o tráfego recebido no porto 80 para o porto 8000 da máquina *host.* Por outro lado, o servidor mitigado só aceita ligações HTTPS, mapeando o seu porto 443 no porto 8001 do *host*. Para este efeito, o servidor contém um certificado X.509 *Secure Sockets Layer*/*Transport Layer Security* (SSL/TLS) auto-assinado – o ficheiro *server.crt* – e a respetiva chave privada RSA – em *server.key*. Para além disto, a única outra diferença entre os dois servidores é o facto de, para permitir a comunicação via HTTPS, o servidor mitigado ter ativado o módulo de SSL. Estas configurações encontram-se nos ficheiros *.htaccess* e *000-default.conf*.

Os dois *sites* seguem a mesma arquitetura lógica, têm o mesmo *sitemap* e utilizam as mesmas tecnologias, sendo a única diferença ao nível da implementação do código e das configurações dos servidores. Ora, sendo este um protótipo *web*, utiliza-se *HyperText Markup Language* (HTML) para estruturar o conteúdo das páginas e *Cascading Style Sheets* (CSS) para as estilizar, bem como PHP para os *scripts* do lado do servidor e *SQLite* como linguagem da base de dados.

A base de dados (cujo esquema se encontra no ficheiro *database.*sql) de cada *site* contém uma única tabela com os utilizadores registados, armazenando o seu nome de utilizador, a respetiva palavra-passe e uma *flag* que indica se o utilizador é, ou não, administrador. O povoamento inicial da base de dados inclui somente um utilizador administrador de cada versão do protótipo: o utilizador *vulnerable* para a versão vulnerável e *mitigated* para a versão mitigada, sendo as respetivas palavras-passe iguais aos próprios nomes.

Cada *site* tem três páginas *web*: a página principal (*index.php*), para efetuar o *login* e o registo dos utilizadores, o perfil de cada utilizador (*profile.php*) e um painel de administração (*admin.php*). A classe *User* – contida no ficheiro *user.php* – rege as interações entre o servidor e a base de dados, realizando as interrogações (*querys*) necessárias.

As ações disponíveis para os utilizadores, realizáveis através da submissão de formulários, são o registo (*register.php*), o início de sessão (*login.php*) e o término da mesma (*logout.php*).

Finalmente, os ficheiros *templates.php* e *style.css* servem, respetivamente, para estruturar o conteúdo mostrado nas páginas e estilizá-lo.

O protótipo inclui ainda dois *scripts* para facilitar a execução do mesmo. O *script start.sh* é responsável por inicializar adequadamente os *containers* e os servidores, incluindo a criação e o povoamento da base de dados (para o ficheiro *database.db* de cada servidor). O *script* presente no ficheiro end*.sh* serve apenas para parar os *containers*, eliminá-los e limpar os ficheiros criados pela execução do *script* anterior.

A01:2021 – Broken Access Control

No contexto da segurança de aplicações *web*, é crucial garantir que os sistemas implementam controlos de acesso rigorosos para proteger informações sensíveis e evitar a exposição indevida de dados, estando protegidos contra ***Broken Access Control***. Duas vulnerabilidades comuns que envolvem falhas neste tipo de controlos são **IDOR** (*Insecure Direct Object Reference*) e ***Forced Browsing***, ambas relacionadas com o acesso não autorizado a recursos que deveriam ser restritos.

A vulnerabilidade de **IDOR** é uma falha que acontece quando uma aplicação *web* não valida adequadamente se o utilizador tem permissão para aceder ou manipular determinados objetos ou recursos, como dados de outros utilizadores ou ficheiros internos. Num ataque de **IDOR**, um atacante pode modificar parâmetros do URL ou de pedidos (como IDs numéricos, por exemplo) para ter acesso a dados que pertencem a outros utilizadores e só deveriam ser acessíveis por eles, mas não estão adequadamente protegidos.

No cenário do *site* vulnerável criado para esta prova de conceito, o sistema utiliza o parâmetro ***id*** para identificar o perfil de utilizador a ser exibido (Figura 1). Contudo, a aplicação não realiza uma validação adequada para garantir que o utilizador que tenta aceder ao URL da Figura 2 tem permissão para tal. Por isso, o perfil do utilizador com *id* igual a 1 ("maria") estará exposto a atacantes que explorem este facto, devido a uma falha de **IDOR**.

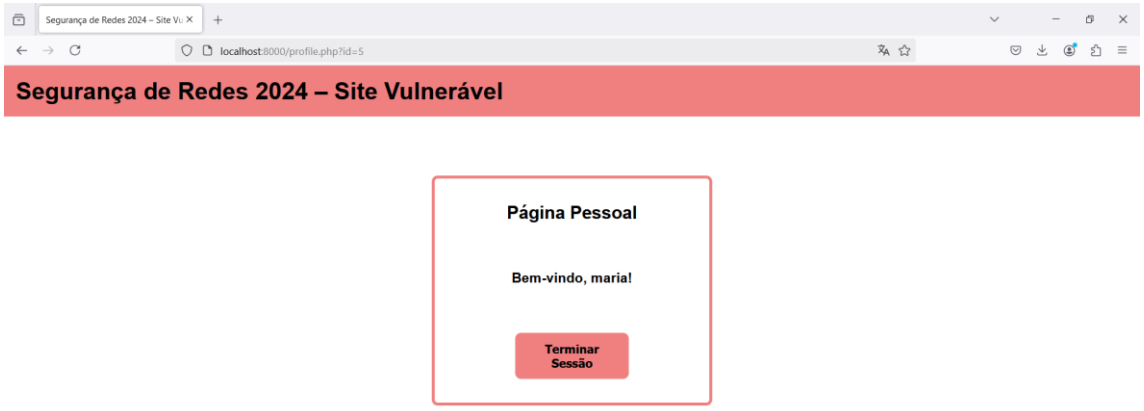


Figura 1 – Página de Perfil do Utilizador "maria"

Figura 2 – URL da Página de Perfil com id=1 *("maria")*

Por outro lado, ***Forced Browsing*** ocorre quando um atacante consegue explorar falhas na configuração de um servidor *web* para aceder a ficheiros ou diretórios sensíveis que não são diretamente acessíveis pela interface da aplicação, ou seja, que não deveriam ser públicos. Estes ataques geralmente acontecem, mais uma vez, por meio da manipulação de URLs ou de caminhos de diretórios. Isto permite que o atacante "force" o navegador a aceder a áreas protegidas, como arquivos de configuração, *logs* ou outros recursos internos. Esta vulnerabilidade pode ser particularmente grave, pois oferece acesso a informações cruciais do sistema, que podem ser utilizadas em ataques mais avançados.

No *site* vulnerável criado para esta prova de conceito, a página de administração da Figura 3, destinada apenas a administradores, está disponível diretamente através do URL descrito na Figura 4. Como a aplicação não realiza verificações adequadas para restringir o acesso apenas a utilizadores autorizados, é possível que um atacante aceda a estes conteúdos confidenciais, bastando, para isso, colocar o URL correto no navegador *web*, realizando um ataque de ***Forced Browsing***.

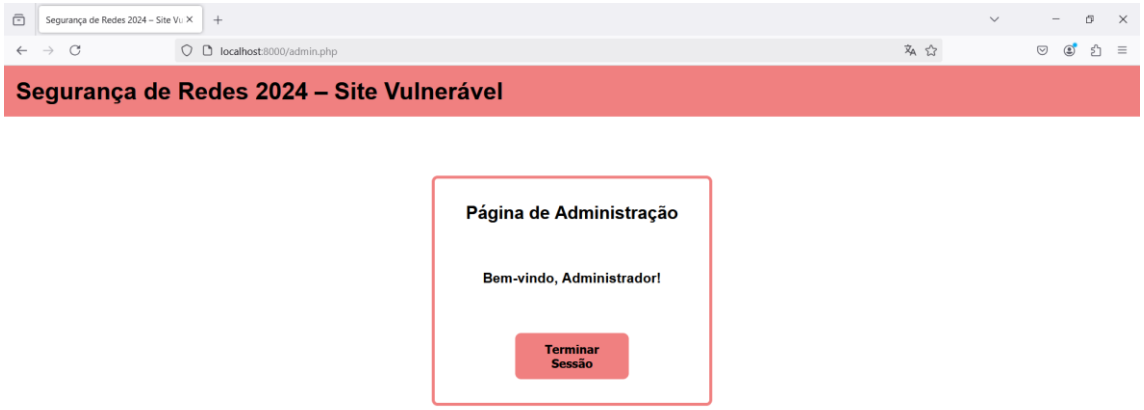


Figura 3 – Página de Administração

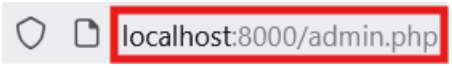


Figura 4 – URL da Página de Administração

Ambas as vulnerabilidades – ***Forced Browsing*** e **IDOR** – têm o potencial de comprometer a confidencialidade, integridade e disponibilidade dos dados de uma aplicação *web*, caso não sejam adequadamente mitigadas.

A contramedida para a vulnerabilidade de **IDOR** consiste em validar adequadamente se o utilizador que está a aceder a uma página *web* pode aceder a essa mesma página. Neste caso, a página vulnerável era o perfil dos utilizadores – *profile.php* –, dado que não existia nenhum mecanismo de controlo de acessos que validasse o acesso à mesma. Efetivamente, era sempre mostrada a página correspondente ao perfil do utilizador cujo ID fosse enviado pelo pedido HTTP GET, ou seja, que estivesse na *query* do URL, associado ao parâmetro *id* (*profile.php?id=…*). Como tal, a contramedida adequada foi adicionar uma dupla verificação para o acesso ao perfil do utilizador, conforme se expõe na Figura 5.

```php
<?php
    declare(strict_types = 1);
    require_once(__DIR__ . '/user.php');
    require_once(__DIR__ . '/templates.php');

    session_set_cookie_params(0, '/', 'localhost', true, true);
    session_start();

    if (!isset($_SESSION['id'])) {
        header("Location: index.php");
        die();
    }

    $db = new PDO('sqlite:' . __DIR__ . '/database.db');
    $db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);

    $id = (int) $_GET['id'];

    if ($id !== $_SESSION['id']) {
        header("Location: profile.php?id=" . $_SESSION['id']);
        die();
    }

    $user = User::getUser($db, $id);

    profile(false, $user);
?>
```

Figura 5 – Contramedida para IDOR

Em primeiro lugar, é verificado se o utilizador que está a tentar aceder ao perfil tem a sessão iniciada no *site*, através da condição do primeiro *if*, que confirma se existe algum valor de ID definido para a sessão atual, procurando-o no *array* da sessão. Se existir, então o utilizador tem sessão iniciada no *site*, mas, se não existir, então o utilizador não está autenticado, pelo que é redirecionado para a página inicial, não conseguindo, assim, aceder indevidamente ao perfil de outro utilizador.

Em segundo lugar – sabendo que o utilizador tem sessão iniciada –, verifica-se se o ID do utilizador cujo perfil se está a tentar aceder (enviado no URL, através do pedido HTTP GET) coincide com o ID do utilizador que tem a sessão iniciada, presente no *array* da sessão. Se coincidir, então é mostrada a página com o perfil do utilizador que, garantidamente, é o próprio que fez o pedido HTTP. Senão, então o utilizador é redirecionado para a página do seu próprio perfil, à qual pode aceder.

Deste modo, estas duas verificações simples asseguram que o controlo de acessos ao perfil de cada utilizador é corretamente efetuado, impedindo, assim, ataques que explorem vulnerabilidades de **IDOR**.

A eficácia desta contramedida é total, dado que impede que um atacante consiga aceder a um perfil sem estar autenticado e, se estiver autenticado, só permite que ele aceda ao seu próprio perfil.

Para além disto, a mitigação contra o ataque de ***Forced Browsing*** é extremamente semelhante, residindo na autenticação correta e no controlo de acessos real e efetivo para as páginas que se pretendem proteger de eventuais atacantes.

No exemplo concreto apresentado, era possível realizar um ataque deste tipo para aceder à página de administração, isto é, a página estava sempre visível/acessível, mesmo para utilizadores não administradores (e até não autenticados). Isto sucedia porque não havia nenhuma validação adequada das permissões do utilizador que tentava aceder à página de administração.

Assim sendo, a contramedida – similar à anterior – consistiu em acrescentar duas verificações necessárias para aceder à página de administração, tal como observável na Figura 6.

```php
<?php
    declare(strict_types = 1);
    require_once(__DIR__ . '/user.php');
    require_once(__DIR__ . '/templates.php');

    session_set_cookie_params(0, '/', 'localhost', true, true);
    session_start();

    if (!isset($_SESSION['id'])) {
        header("Location: index.php");
        die();
    }

    $db = new PDO('sqlite:' . __DIR__ . '/database.db');
    $db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);

    $id = $_SESSION['id'];
    $user = User::getUser($db, $id);

    if (!$user->isAdmin()) {
        header("Location: profile.php?id=" . $user->getId());
        die();
    }

    admin(false);
?>
```

Figura 6 – Contramedida para Forced Browsing

A primeira verificação adicionada é exatamente igual à já explicada anteriormente, tendo como efeito a confirmação de que o utilizador que está a tentar aceder à página de administração tem, efetivamente, a sessão corretamente iniciada no *site*, assegurada através da existência de um *id* da sessão. Se esta verificação falhar, o utilizador é redirecionado para a página principal. Senão, o servidor continua o processamento para a próxima verificação.

A segunda verificação – só efetuada quando o utilizador já tem sessão iniciada – consiste em verificar se o utilizador tem privilégios de administração, ou seja, se é administrador, através da função `isAdmin()`. Esta função, definida na classe *User* (no ficheiro *user.php*) efetua uma interrogação à base de dados para verificar o valor armazenado para o atributo *admin* do utilizador correspondente. Se o utilizador tiver este valor definido como 1, então é administrador, pelo que pode aceder à página de administração. Caso contrário, o utilizador autenticado não tem privilégios de administração, de modo que, como não pode visualizar o painel exclusivo a administradores, é redirecionado para a página do seu perfil.

De forma análoga à anterior, esta contramedida também é totalmente eficaz contra qualquer tipo de ataques de ***Forced Browsing***, visto que acrescenta os controlos de segurança em falta para mitigar a falha de *Broken Access Control*, forçando, assim, a que só os administradores tenham acesso à página de administração.

A02:2021 – Cryptographic Failures

As ***Cryptographic Failures*** (falhas criptográficas) representam uma das vulnerabilidades mais críticas em sistemas de informação, ocorrendo quando algoritmos criptográficos ou as suas implementações não são utilizadas adequadamente para proteger dados sensíveis. Exemplos comuns incluem o uso dos algoritmos de *hash* **MD5** ou **SHA-1** (considerados inseguros por já terem sido quebrados), armazenar senhas cifradas sem *salt*, ou a troca de informações críticas sem o uso de protocolos seguros como HTTPS. Estas falhas podem comprometer a confidencialidade, integridade e autenticidade das informações, expondo-as a manipulação ou roubo por parte de atacantes.

Uma das medidas de prevenção para ataques deste tipo passa por classificar os dados processados, armazenados ou transmitidos pela aplicação e identificar quais desses dados são considerados sensíveis. Posto isto, deve evitar-se armazenar desnecessariamente dados sensíveis e garantir a sua encriptação, para que, caso a aplicação *web* esteja vulnerável, estes dados não sejam comprometidos.

Efetivamente, demonstra-se que o *site* vulnerável criado não só expõe a sua base de dados, acessível pelo URL da Figura 7, como também não garante a encriptação segura da mesma e dos dados nela armazenados.

Figura 7 – URL da Base de Dados

Obtida a base de dados associada ao *site* vulnerável, torna-se possível explorá-la através da utilização de ***SQLite***, tal como representado na Figura 8. Inicialmente, ao executar o comando **`.tables`**, identifica-se a existência de uma tabela chamada ***users***, que armazena informações dos utilizadores do sistema, sendo esta a única tabela presente na base de dados.

Em seguida, com o comando **`PRAGMA table_info(users);`**, é observável a estrutura da tabela, concluindo-se que esta possui as colunas ***id***, ***username***, ***password*** (*hashed*) e ***admin*** (um indicador de privilégios de administração). Finalmente, ao executar o comando **`SELECT * FROM users;`**, são expostos os dados armazenados na tabela.

```
SQLite version 3.37.2 2022-01-06 13:25:41
Enter ".help" for usage hints.
sqlite> .tables
users
sqlite> PRAGMA table_info(users);
0|id|INTEGER|0||1
1|username|VARCHAR(255)|1||0
2|password|VARCHAR(255)|1||0
3|admin|INTEGER|0|0|0
sqlite> SELECT * FROM users;
1|vulnerable|25890deab1075e916c06b9e1efc2e25f|1
2|mitigated|$2y$12$LIqlGBmRo3fE1iZZCcWrROyAb7gPqJPiOopk9frLtL0li7.WfuD72|1
3|matilde|641a3af3fd54d3a81d47e23376dcc90a|0
4|manel|95ab6cdfaa9d646f83061a936bfb8996|0
5|maria|263bce650e68ab4e23f28263760b9fa5|0
```

Figura 8 – Exploração da Base de Dados com SQLite

Assim, utilizando ferramentas como ***hash-identifier*** (para identificar possíveis tipos de *hash*), ***hashcat*** ou ***CrackStation*** (Figura 9), as senhas armazenadas na base de dados da versão vulnerável são totalmente recuperáveis. Isto é possível porque as palavras-passe são cifradas com um algoritmo criptográfico fraco e já previamente quebrado – MD5 – e não são guardadas com *salt*, contrariando, deste modo, as boas práticas no que diz respeito ao armazenamento de palavras-passe, tornando-as vulneráveis.

Hash	Type	Result
25890deab1075e916c06b9e1efc2e25f	md5	vulnerable
641a3af3fd54d3a81d47e23376dcc90a	md5	matilde
95ab6cdfaa9d646f83061a936bfb8996	md5	manel
263bce650e68ab4e23f28263760b9fa5	md5	maria

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

Figura 9 – Utilização da Ferramenta CrackStation para Quebrar as Hashes

A falha identificada no código da aplicação *web* vulnerável está relacionada com o uso indevido de algoritmos criptográficos para guardar e verificar palavras-passe. O algoritmo MD5 utilizado – tal como é visível na função de registo da Figura 10 – é amplamente reconhecido como inseguro devido ao facto de ser vulnerável a ataques de colisão e força bruta, especialmente quando não é utilizado com medidas adicionais, como o *salt*. Esta abordagem compromete a confidencialidade das credenciais, expondo os utilizadores a riscos significativos se algum atacante conseguir aceder indevidamente à base de dados.

```
public static function register(PDo $db, string $username, string
$password) : ?User {
    $stmt = $db->prepare('
        INSERT INTO users (username, password)
        VALUES (?, ?)
    ');

    try {
        $stmt->execute(array(strtolower($username), md5($password)));
    } catch (PDOException $e) {
        return null;
    }

    return User::login($db, $username, $password);
}
```

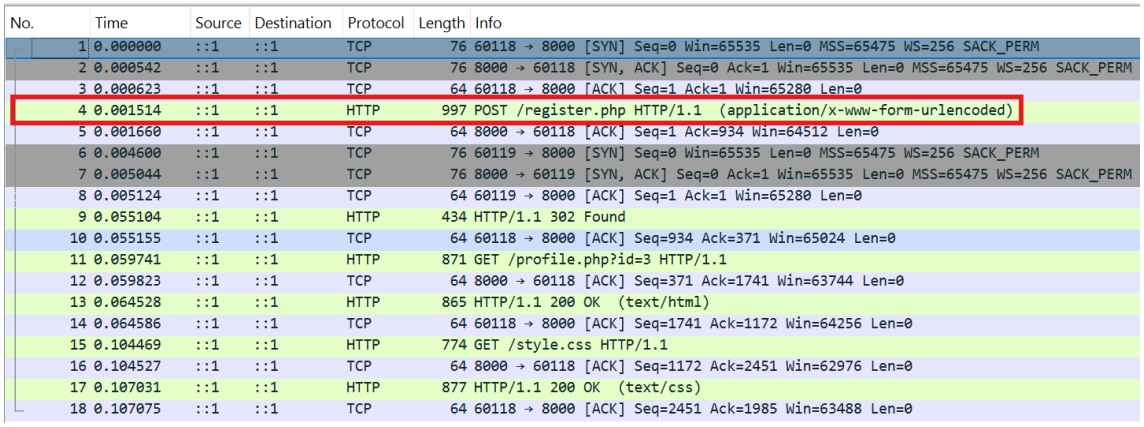
Figura 10 – Registo Vulnerável

Analogamente, na função de autenticação exposta na Figura 11, o algoritmo MD5 é utilizado para verificar se a *hash* armazenada na base de dados corresponde à *hash* da palavra-passe introduzida pelo utilizador na tentativa de *login*.

```
public static function login(PDO $db, string $username, string
$password) : ?User {
    $stmt = $db->prepare('
        SELECT id, username, password, admin
        FROM users
        WHERE lower(username) = ?
    ');

    $stmt->execute(array(strtolower($username)));
    $user = $stmt->fetch();

    if ($user && md5($password) === $user['password'])
        return new User((int) $user['id'], $user['username'], (bool)
$user['admin']);

    return null;
}
```

Figura 11 – Autenticação Vulnerável

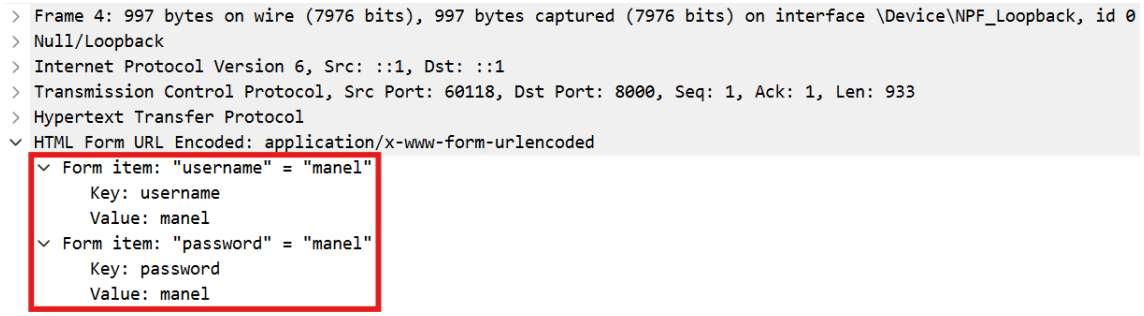
Para além disto, o servidor vulnerável só aceita conexões HTTP e troca informações sensíveis através deste protocolo de comunicação, logo, não encriptadas e, por isso, visíveis para um atacante (*Man-in-the-Middle*). Tal como é possível observar na Figura 12, um *eavesdropper* que esteja à escuta no canal de comunicação consegue capturar o pacote em que o utilizador efetua o registo no *site* vulnerável, conseguindo, deste modo, obter as suas credenciais de forma indevida.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	::1	::1	TCP	76	60118 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
2	0.000542	::1	::1	TCP	76	8000 → 60118 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
3	0.000623	::1	::1	TCP	64	60118 → 8000 [ACK] Seq=1 Ack=1 Win=65280 Len=0
4	0.001514	::1	::1	HTTP	997	POST /register.php HTTP/1.1 (application/x-www-form-urlencoded)
5	0.001660	::1	::1	TCP	64	8000 → 60118 [ACK] Seq=1 Ack=934 Win=64512 Len=0
6	0.004600	::1	::1	TCP	76	60119 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
7	0.005044	::1	::1	TCP	76	8000 → 60119 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
8	0.005124	::1	::1	TCP	64	60119 → 8000 [ACK] Seq=1 Ack=1 Win=65280 Len=0
9	0.055104	::1	::1	HTTP	434	HTTP/1.1 302 Found
10	0.055155	::1	::1	TCP	64	60118 → 8000 [ACK] Seq=934 Ack=371 Win=65024 Len=0
11	0.059741	::1	::1	HTTP	871	GET /profile.php?id=3 HTTP/1.1
12	0.059823	::1	::1	TCP	64	8000 → 60118 [ACK] Seq=371 Ack=1741 Win=63744 Len=0
13	0.064528	::1	::1	HTTP	865	HTTP/1.1 200 OK (text/html)
14	0.064586	::1	::1	TCP	64	60118 → 8000 [ACK] Seq=1741 Ack=1172 Win=64256 Len=0
15	0.104469	::1	::1	HTTP	774	GET /style.css HTTP/1.1
16	0.104527	::1	::1	TCP	64	8000 → 60118 [ACK] Seq=1172 Ack=2451 Win=62976 Len=0
17	0.107031	::1	::1	HTTP	877	HTTP/1.1 200 OK (text/css)
18	0.107075	::1	::1	TCP	64	60118 → 8000 [ACK] Seq=2451 Ack=1985 Win=63488 Len=0

Figura 12 – Pacote de Registo

Como a comunicação não é encriptada, o pacote em questão contém a informação enviada pelo cliente ao servidor em *plaintext*, como é visível na Figura 13.



```
> Frame 4: 997 bytes on wire (7976 bits), 997 bytes captured (7976 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 6, Src: ::1, Dst: ::1
> Transmission Control Protocol, Src Port: 60118, Dst Port: 8000, Seq: 1, Ack: 1, Len: 933
> Hypertext Transfer Protocol
v HTML Form URL Encoded: application/x-www-form-urlencoded
  v Form item: "username" = "manel"
      Key: username
      Value: manel
  v Form item: "password" = "manel"
      Key: password
      Value: manel
```

Figura 13 – Credenciais de Registo do Utilizador

Na versão mitigada do código, foram realizadas alterações importantes para resolver os riscos associados às falhas criptográficas.

Em primeiro lugar, foi utilizada a função *password_hash* com o algoritmo *bcrypt* para guardar as palavras-passe. Este algoritmo é fortemente recomendado devido à sua robustez e eficácia contra ataques de força bruta, além de incorporar automaticamente um *salt* único e aleatório para cada palavra-passe, protegendo contra ataques de *rainbow tables*.

Adicionalmente, foi configurada uma opção personalizada de custo computacional (*cost* de 12) na função *password_hash*. Esta configuração aumenta o tempo de processamento necessário para gerar e verificar as *hashes*, tornando os ataques de força bruta significativamente mais dispendiosos computacionalmente.

Todas estas alterações são visíveis na função de registo presente na Figura 14.

```
public static function register(PDO $db, string $username, string
$password) : ?User {
    $stmt = $db->prepare('
        INSERT INTO users (username, password)
        VALUES (?, ?)
    ');

    $options = ['cost' => 12];
    $hashedPassword = password_hash($password, PASSWORD_BCRYPT,
$options);

    try {
        $stmt->execute(array(strtolower($username), $hashedPassword));
    } catch (PDOException $e) {
        return null;
    }

    return User::login($db, $username, $password);
}
```

Figura 14 – Registo Mitigado

Para garantir consistência e correção, a função *password_verify* foi utilizada na função de autenticação exposta na Figura 15, garantindo que a palavra-passe fornecida pelo utilizador é comparada com a *hash* guardada de forma segura e resistente a ataques de *timing* (em que o atacante consegue extrair informações sobre a senha com base no tempo em que demora a comparação da palavra-passe introduzida com a armazenada na base de dados), já contemplando/tratando também o *salt*.

```
public static function login(PDO $db, string $username, string
$password) : ?User {
    $stmt = $db->prepare('
        SELECT id, username, password, admin
        FROM users
        WHERE lower(username) = ?
    ');

    $stmt->execute(array(strtolower($username)));
    $user = $stmt->fetch();

    if ($user && password_verify($password, $user['password']))
        return new User((int) $user['id'], $user['username'], (bool)
$user['admin']);

    return null;
}
```

Figura 15 – Autenticação Mitigada

Além disto, de maneira a resolver a falha que surgia pela falta de encriptação nas comunicações entre cliente e servidor, o protocolo HTTP foi substituído por HTTPS. Para tal, foi necessário criar um certificado SSL/TLS auto-assinado, bem como a respetiva chave privada. A par disto, foram alteradas as configurações do servidor para obrigar a que todas as comunicações sejam feitas por HTTPS, mitigando a falha criptográfica. Os ficheiros modificados expõem-se nas Figuras 16 e 17.

```
<VirtualHost *:443>
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

    SSLEngine on
    SSLCertificateFile /etc/ssl/certs/server.crt
    SSLCertificateKeyFile /etc/ssl/private/server.key

    <Directory /var/www/html>
        AllowOverride All
    </Directory>
</VirtualHost>
```

Figura 16 – Ficheiro 000-default.conf

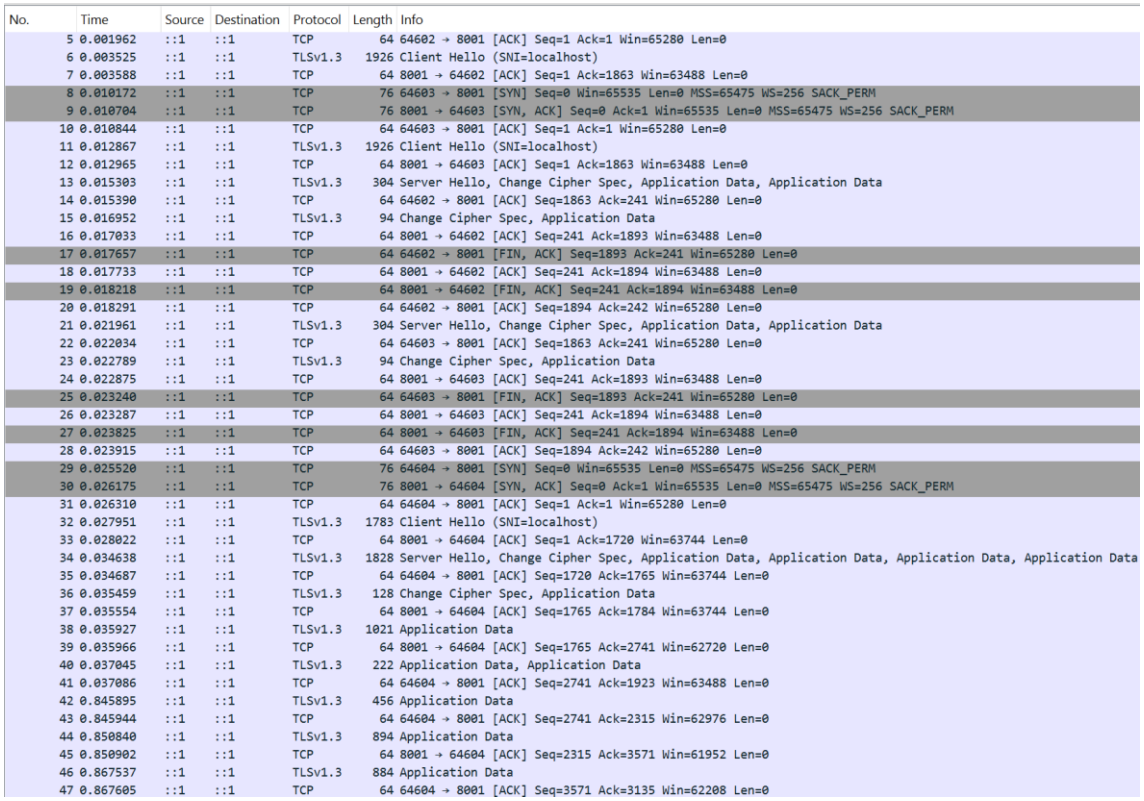
O ficheiro *000-default.conf* (Figura 16) ativa o módulo SSL no servidor, localizando os ficheiros com o certificado do servidor e a chave privada correspondente, criados previamente através do comando **`openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout server.key -out server.crt`**.

```
RewriteEngine On

RewriteRule ^users/([0-9]+)$ profile.php?id=$1 [L,QSA]

RewriteCond %{HTTPS} !=on
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

Figura 17 – Ficheiro .htaccess

O ficheiro *.htaccess* (Figura 17) obriga a que qualquer cliente que tente comunicar com o servidor tenha de o fazer por HTTPS, para garantir confidencialidade.

Deste modo, tal como é possível verificar na Figura 18, o pedido de registo efetuado no *site* mitigado encripta o conteúdo enviado no formulário, impossibilitando a recuperação do mesmo por parte de um atacante. Assim, um *eavesdropper* consegue perceber que foi trocada informação entre o cliente e o servidor, mas, como esta está encriptada, não consegue perceber qual é o seu conteúdo.



No.	Time	Source	Destination	Protocol	Length	Info
5	0.001962	::1	::1	TCP	64	64602 → 8001 [ACK] Seq=1 Ack=1 Win=65280 Len=0
6	0.003525	::1	::1	TLSv1.3	1926	Client Hello (SNI=localhost)
7	0.003588	::1	::1	TCP	64	8001 → 64602 [ACK] Seq=1 Ack=1863 Win=63488 Len=0
8	0.010172	::1	::1	TCP	76	64603 → 8001 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
9	0.010704	::1	::1	TCP	76	8001 → 64603 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
10	0.010844	::1	::1	TCP	64	64603 → 8001 [ACK] Seq=1 Ack=1 Win=65280 Len=0
11	0.012867	::1	::1	TLSv1.3	1926	Client Hello (SNI=localhost)
12	0.012965	::1	::1	TCP	64	8001 → 64603 [ACK] Seq=1 Ack=1863 Win=63488 Len=0
13	0.015303	::1	::1	TLSv1.3	304	Server Hello, Change Cipher Spec, Application Data, Application Data
14	0.015390	::1	::1	TCP	64	64602 → 8001 [ACK] Seq=1863 Ack=241 Win=65280 Len=0
15	0.016952	::1	::1	TLSv1.3	94	Change Cipher Spec, Application Data
16	0.017033	::1	::1	TCP	64	8001 → 64602 [ACK] Seq=241 Ack=1893 Win=63488 Len=0
17	0.017657	::1	::1	TCP	64	64602 → 8001 [FIN, ACK] Seq=1893 Ack=241 Win=65280 Len=0
18	0.017733	::1	::1	TCP	64	8001 → 64602 [ACK] Seq=241 Ack=1894 Win=63488 Len=0
19	0.018218	::1	::1	TCP	64	8001 → 64602 [FIN, ACK] Seq=241 Ack=1894 Win=63488 Len=0
20	0.018291	::1	::1	TCP	64	64602 → 8001 [ACK] Seq=1894 Ack=242 Win=65280 Len=0
21	0.021961	::1	::1	TLSv1.3	304	Server Hello, Change Cipher Spec, Application Data, Application Data
22	0.022034	::1	::1	TCP	64	64603 → 8001 [ACK] Seq=1863 Ack=241 Win=65280 Len=0
23	0.022789	::1	::1	TLSv1.3	94	Change Cipher Spec, Application Data
24	0.022875	::1	::1	TCP	64	8001 → 64603 [ACK] Seq=241 Ack=1893 Win=63488 Len=0
25	0.023240	::1	::1	TCP	64	64603 → 8001 [FIN, ACK] Seq=1893 Ack=241 Win=65280 Len=0
26	0.023287	::1	::1	TCP	64	8001 → 64603 [ACK] Seq=241 Ack=1894 Win=63488 Len=0
27	0.023825	::1	::1	TCP	64	8001 → 64603 [FIN, ACK] Seq=241 Ack=1894 Win=63488 Len=0
28	0.023915	::1	::1	TCP	64	64603 → 8001 [ACK] Seq=1894 Ack=242 Win=65280 Len=0
29	0.025520	::1	::1	TCP	76	64604 → 8001 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
30	0.026175	::1	::1	TCP	76	8001 → 64604 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM
31	0.026310	::1	::1	TCP	64	64604 → 8001 [ACK] Seq=1 Ack=1 Win=65280 Len=0
32	0.027951	::1	::1	TLSv1.3	1783	Client Hello (SNI=localhost)
33	0.028022	::1	::1	TCP	64	8001 → 64604 [ACK] Seq=1 Ack=1720 Win=63744 Len=0
34	0.034638	::1	::1	TLSv1.3	1828	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
35	0.034687	::1	::1	TCP	64	64604 → 8001 [ACK] Seq=1720 Ack=1765 Win=63744 Len=0
36	0.035459	::1	::1	TLSv1.3	128	Change Cipher Spec, Application Data
37	0.035554	::1	::1	TCP	64	8001 → 64604 [ACK] Seq=1765 Ack=1784 Win=63744 Len=0
38	0.035927	::1	::1	TLSv1.3	1021	Application Data
39	0.035966	::1	::1	TCP	64	8001 → 64604 [ACK] Seq=1765 Ack=2741 Win=62720 Len=0
40	0.037045	::1	::1	TLSv1.3	222	Application Data, Application Data
41	0.037086	::1	::1	TCP	64	64604 → 8001 [ACK] Seq=2741 Ack=1923 Win=63488 Len=0
42	0.845895	::1	::1	TLSv1.3	456	Application Data
43	0.845944	::1	::1	TCP	64	64604 → 8001 [ACK] Seq=2741 Ack=2315 Win=62976 Len=0
44	0.850840	::1	::1	TLSv1.3	894	Application Data
45	0.850902	::1	::1	TCP	64	8001 → 64604 [ACK] Seq=2315 Ack=3571 Win=61952 Len=0
46	0.867537	::1	::1	TLSv1.3	884	Application Data
47	0.867605	::1	::1	TCP	64	64604 → 8001 [ACK] Seq=3571 Ack=3135 Win=62208 Len=0

Figura 18 – Comunicação por HTTPS

Estas alterações asseguram que as credenciais dos utilizadores estão protegidas conforme as melhores práticas atuais e recomendadas pelo OWASP, resolvendo eficazmente as falhas criptográficas do código vulnerável. Demonstrou-se a importância de escolher algoritmos criptográficos seguros e atuais, bem como de configurar corretamente os parâmetros de segurança e assegurar a encriptação da comunicação entre cliente e servidor, para garantir a proteção das credenciais dos utilizadores.

Conclusão

Em suma, através desta prova de conceito foram explorados e demonstrados alguns ataques possíveis que exploram falhas abrangidas pelas duas categorias mais comuns do OWASP *Top 10* – ***Broken Access Control*** e ***Cryptographic Failures*** –, bem como implementadas as respetivas contramedidas e avaliada a sua eficácia.

Deste modo, foi possível verificar que, na verdade, a prevenção e mitigação de ataques deste tipo não é um processo extremamente complexo, sendo apenas necessário seguir as boas práticas no que diz respeito ao controlo de acessos e à segurança criptográfica. Para além disso, ficou também patente a facilidade com que um atacante pode conseguir explorar qualquer falha de segurança nestes dois domínios, com potenciais impactos ao nível da confidencialidade e integridade dos sistemas *web*.

Assim sendo, conclui-se que os objetivos do trabalho foram cumpridos.