INSTITUTO FEDERAL DO PARANÁ - CAMPUS PINHAIS

CURSO SUPERIOR DE TECNOLOGIA EM GESTÃO DA TECNOLOGIA DA INFORMAÇÃO

PROJETO INTERDISCIPLINAR III

Segurança Essencial em PHP

Protegendo Dados em Projetos de Impacto Social

Autor: RODRIGO DE LIMA
Revisado: MARCELO A MACIEL JR

PINHAIS
2025

Sumário

2-3

Introdução: O Contrato de Confiança

A Responsabilidade por Trás do Código..... 4
Nosso Mapa da Jornada..... 5

Capítulo 1: A Chave do Castelo: Protegendo o Acesso Principal

O Erro que Muitos Cometem com Senhas..... 6
A Solução Moderna do PHP: password_hash()..... 7
Mão na Massa - Salvando a Senha no Cadastro..... 8-9
Mão na Massa - Verificando a Senha no Login..... 9-10-11

Capítulo 2: O Crachá Mágico: Mantendo o Usuário Conectado com Segurança

O Problema: Como um Site "Lembra" de Você?..... 12
Entregando o Crachá Após o Login..... 13-14
Verificando o Crachá em Cada Sala..... 15-16
Saindo do Evento: O Logout Seguro..... 16-17-18

Capítulo 3: O Porteiro Atento: Filtrando Dados Maliciosos

SQL Injection - O Ataque do "Cavalo de Troia"..... 19-20
A defesa: Prepared Statements.....20-21-22
Cross-Site Scripting (XSS) - A "Pichação" Digital..... 22,23
A Defesa: Limpando o que Será Exibido..... 23-24

PINHAIS
2025

Capítulo 4: O Manual do Respeito: A LGPD na Prática

LGPD - O Que o Desenvolvedor Precisa Saber?..... 25-26
O Direito ao Esquecimento - O Fluxo da Exclusão..... 26-27
Mão na Massa - O Código por Trás do "Apagar Tudo"..... 27-28-29-30

Próximos Passos: Da Teoria à Prática Contínua

Checklist Final de Segurança..... 31-32
Mantendo a Fortaleza Segura: Boas Práticas Contínuas.....32
Protegendo o Terreno: Segurança Fora do Código PHP..... .32-33
Guardando o Mapa do Tesouro em um Lugar Secreto..... 34-35

Conclusão

O Código Como um Ato de Cuidado..... 36

Referências

.....37-38

Glossário

.....39-40-41

PINHAIS
2025

A Responsabilidade por Trás do Código

O Peso de Um Projeto Social

Imagine que você não está apenas escrevendo código, mas construindo uma casa. Não uma casa qualquer, mas um centro de apoio, um lugar seguro para uma comunidade que precisa de ajuda e acolhimento. Neste lugar, as paredes são o seu código, a porta de entrada é a tela de login, e os registros guardados nos armários são os dados dos usuários.

A primeira coisa que você garante em uma construção física é um alicerce sólido e paredes fortes, certo? No mundo digital, esse alicerce tem um nome: segurança. A confiança que as pessoas depositam em um projeto social não é automática; ela é construída sobre a promessa de que suas informações, muitas vezes sensíveis e pessoais, estarão protegidas.

É aqui que a nossa responsabilidade como desenvolvedores se torna imensa. Se um formulário de cadastro é vulnerável, não é apenas uma "falha no sistema". É como deixar a porta do centro de apoio destrancada durante a noite. Se senhas são vazadas, não é só um "incidente técnico". É como perder a chave do arquivo onde estão guardados os prontuários e as histórias de vida das pessoas que confiam em você.

Por isso, antes de escrevermos a primeira linha de código sobre funcionalidades, precisamos falar sobre proteção. Este guia foi pensado para isso: para te mostrar que a segurança não é um "extra" ou um recurso avançado. **É a base, o contrato de confiança que firmamos com cada pessoa que decide usar nossa plataforma.** Vamos aprender a construir paredes fortes e a trancar as portas do jeito certo.

Nosso Mapa da Jornada

O Que Você Vai Aprender Neste Guia?

Agora que entendemos a importância de construir um alicerce seguro, vamos olhar para o mapa que guiará nossa construção. A segurança digital pode parecer um labirinto complexo, mas nosso objetivo aqui é traçar um caminho reto e claro pelos pontos mais essenciais.

Ao final desta jornada, você terá o conhecimento prático para implementar as quatro camadas fundamentais de proteção em seus projetos PHP:

- A Chave do Castelo: Aprenderemos a forma correta e moderna de armazenar senhas, garantindo que elas fiquem ilegíveis até mesmo para quem administra o banco de dados.
- O Crachá Mágico: Vamos construir um sistema de login que "lembra" do usuário autenticado de forma segura, protegendo páginas e recursos que não devem ser públicos.
- O Porteiro Atento: Descobriremos como blindar os formulários do seu site contra os dois tipos de ataques mais comuns na web, que tentam enganar seu sistema para roubar informações.
- O Manual do Respeito: Entenderemos como colocar em prática o respeito à privacidade do usuário, implementando funcionalidades essenciais exigidas pela Lei Geral de Proteção de Dados (LGPD).

Para quem é este guia?

Este material foi pensado e escrito para qualquer pessoa que deseje construir aplicações web mais seguras e responsáveis. Você aproveitará ao máximo se for:

- Um(a) estudante de tecnologia que está dando os primeiros passos em programação.
- Um(a) desenvolvedor(a) iniciante em PHP que busca construir um portfólio sólido e seguro.
- Um(a) criador(a) de projetos (sociais ou não) que lidará com dados de usuários e quer começar com o pé direito, fazendo da segurança uma prioridade.

Capítulo 1: A Chave do Castelo: Protegendo o Acesso Principal

1.1 O Erro que Muitos Cometem com Senhas.

1.1.1 - Por que "Embaralhar" Senhas do Jeito Antigo não Funciona Mais

Toda vez que um usuário se cadastra em nosso site, ele nos confia uma das suas chaves mais importantes: a senha. O nosso primeiro dever é guardar essa chave de uma forma que ninguém, nem mesmo nós, os administradores do sistema, consiga vê-la em sua forma original.

Para fazer isso, usamos uma técnica chamada hash.

Pense em um *hash* como o resultado de um liquidificador. Você coloca uma fruta dentro dele (a senha, por exemplo, "senha123"), liga, e o que sai é uma vitamina (o hash, um código longo e complexo como e8666d6705c5a20822b5e23c72b5350b). A mágica do liquidificador é que ele é uma via de mão única: é impossível pegar a vitamina e transformá-la de volta na fruta original. Com o hash, a ideia é a mesma.

Por muitos anos, programadores usaram "liquidificadores" mais antigos, como um algoritmo chamado MD5. O problema é que esses modelos antigos eram muito simples.

Com o tempo, hackers criaram "catálogos" gigantescos (conhecidos como *rainbow tables*) que continham milhões de frutas já batidas. Para descobrir a senha original, bastava pegar a "vitamina" (o hash) e procurar nesse catálogo qual "fruta" a gerou. Era como usar um liquidificador de vidro: todo mundo conseguia ver o que estava acontecendo lá dentro.

Hoje, usar MD5 para senhas é como tentar proteger um tesouro com um cadeado de papel. Simplesmente não funciona mais. Precisamos de um método mais forte, um verdadeiro cofre digital. E a boa notícia é que o PHP nos oferece um, pronto para usar.

1.2- A Solução Moderna do PHP: password_hash()

1.2.1 - O Cofre Digital do PHP

Se os métodos antigos são como liquidificadores de vidro, a solução que o PHP nos oferece hoje é como um cofre de titânio com segredo aleatório. O nome dessa solução é uma função chamada `password_hash()`.

O que torna essa função tão especial e segura? Ela faz duas coisas de forma automática que a tornam infinitamente superior aos métodos antigos:

1. Usa um "Liquidificador" de Última Geração: Por padrão, a função utiliza algoritmos de hash modernos e extremamente fortes (como o *Bcrypt*). Eles foram projetados para serem lentos de propósito. Isso pode parecer ruim, mas é uma característica genial: se é lento para o seu sistema verificar uma senha, imagine o quão insuportavelmente lento seria para um hacker tentar testar bilhões de combinações!
2. Adiciona um "Ingrediente Secreto" (Salt): Esta é a parte mais inteligente. Antes de "bater" a senha no liquidificador, a função password_hash() adiciona um "ingrediente secreto" aleatório, único para cada senha. Esse ingrediente é chamado de salt (sal, em inglês).

Isso significa que se dois usuários diferentes tiverem a mesma senha, como "123456", o resultado final (o hash) guardado no banco de dados será completamente diferente para cada um. O "catálogo de vitaminas prontas" dos hackers se torna inútil, pois eles nunca saberão qual foi o ingrediente secreto usado.

A melhor parte é que você não precisa se preocupar com toda essa complexidade. O PHP cuida de tudo: ele escolhe o melhor algoritmo, gera o "salt" aleatório e entrega o hash final prontinho para ser guardado. Nosso único trabalho é chamar a função certa. Vamos ver como fazer isso na prática.

1.3- Mão na Massa- Salvando a Senha no Cadastro

1.3.1 - Guardando a Chave no Cofre

Vamos imaginar a cena: um novo usuário acabou de preencher o formulário de cadastro em nosso site e clicou em "Enviar". As informações, incluindo a senha que ele escolheu, chegam ao nosso servidor. É neste exato momento que precisamos agir para proteger essa senha antes de guardá-la.
O código para fazer isso é surpreendentemente simples. Veja abaixo como seria um trecho do nosso arquivo PHP que processa o cadastro:

PHP

```
<?php

PASSO 1: Pegar a senha que o usuário digitou no formulário.
A variável $_POST['senha'] busca o que foi escrito no campo com

name="senha" lá no HTML.
$senha_original = $_POST['senha'];


PASSO 2: Criptografar a senha para guardá-la de forma segura.
Esta é a parte mais importante para a segurança.
Usamos a função password_hash para transformar a senha em um
código "embaralhado" (hash).

O PASSWORD_DEFAULT garante que o PHP sempre use o método de
criptografia mais forte do momento.

$senha_segura = password_hash($senha_original, PASSWORD_DEFAULT);

PASSO 3: Guardar a senha já criptografada no banco de dados.

 A variável $senha_segura agora contém o código protegido, e não
a senha real.
É este código que você deve salvar na coluna de senha da sua
tabela de usuários.

Exemplo de como ficaria o comando para inserir no banco:
```

```
INSERT INTO usuarios (nome, email, senha_hash) VALUES ('Maria
Silva', 'maria@email.com', '$senha_segura');

Atenção: A senha original, que estava na variável $senha_original,
nunca é salva.
Depois de criptografada, ela não é mais usada.
 ?>
```

Analisando o código:

Repare como o processo é direto. Em apenas uma linha de código, transformamos uma senha vulnerável ("senha123") em um hash completamente seguro (algo como $2y$10$... etc.).

A senha original do usuário nunca toca o banco de dados. Nós guardamos apenas o resultado "embaralhado" e seguro. Com isso, a chave do nosso usuário está agora trancada dentro do cofre digital.

Mas como fazemos para abrir a porta depois? Vamos ver a seguir.

1.4- Mão na Massa- Verificando a Senha no Login

1.4.1 - Usando a Chave para Abrir a Porta

Agora que a senha do nosso usuário está seguramente guardada no banco de dados, precisamos de um jeito de verificar se a senha que ele digita na tela de login é a correta.

Lembre-se: o hash é uma via de mão única. Não podemos "desembaralhar" o código que salvamos para compará-lo com a senha digitada. Então, como fazemos?

É simples: o PHP nos dá uma função "parceira" do senha_hash(), feita exatamente para isso. O nome dela é verificar a senha ().

O processo é como usar uma chave mestra:

1. O usuário digita o e-mail e a senha no formulário de login.
2. Nós usamos o e-mail para encontrar o registro daquele usuário no banco de dados e pegar o hash que guardamos durante o cadastro.
3. Entregamos para a função password_verify() as duas coisas: a senha que o usuário acabou de digitar e o hash que estava guardado.

A função faz toda a mágica por trás dos panos e nos responde com um simples "sim" (verdadeiro) ou "não" (falso).

Veja como o código é limpo e seguro:

PHP

```
<?php

 PASSO 1: PEGAR OS DADOS DO FORMULÁRIO DE LOGIN

$usuario_email = $_POST['email'];      e-mail que foi digitado
$usuario_senha = $_POST['senha'];     senha que foi digitada

PASSO 2: PROCURAR O USUÁRIO NO BANCO DE DADOS
Aqui você normalmente faria uma consulta no banco de dados
usando o e-mail digitado para achar o registro do usuário.

Exemplo de consulta SQL:
SELECT id, nome, senha_hash FROM usuarios WHERE email =
'$usuario_email';

/Vamos imaginar que o banco retornou o hash da senha guardado
assim:
$senha_guardada_no_banco =
'$2y$10$exemploDeHashSalvoNoSeuBancoDeDados';

PASSO 3: VERIFICAR SE A SENHA DIGITADA ESTÁ CORRETA

A função password_verify compara a senha digitada com o hash
salvo.
 Ela retorna TRUE se a senha estiver correta, ou FALSE se estiver
errada.
if (password_verify($usuario_senha, $senha_guardada_no_banco)) {

 Se chegou aqui, a senha está certa!
    echo "✅ Login realizado com sucesso! Bem-vindo(a)!";

     Aqui você poderia iniciar a sessão do usuário, por exemplo:
     session_start();
    $_SESSION['usuario'] = $usuario_email;

} else {
```

```
    Se chegou aqui, a senha está errada.
    echo "❌ E-mail ou senha incorretos. Tente novamente.";

}

?>
```

Com isso, fechamos o ciclo de autenticação de forma segura. Nunca expomos a senha original e usamos as melhores práticas que o PHP oferece. Agora, estamos prontos para manter o usuário conectado.

Capítulo 2: O Crachá Mágico: Mantendo o Usuário Conectado com Segurança.

2.1- O Problema: Como um Site "Lembra" de Você?

2.1.1 - A Memória Curta da Internet

Conseguimos fazer nosso usuário entrar no sistema de forma segura. Ótimo! Mas agora enfrentamos um novo desafio. Por natureza, a internet tem "memória curta". Cada vez que você clica em um link e navega para uma nova página, o servidor web te trata como um completo estranho. Ele não tem ideia de quem você é ou do que fez na página anterior.

Imagine a seguinte situação: o usuário faz o login na página inicial e clica no link "Meus Agendamentos". Quando a página de agendamentos carrega, o servidor pensa: "Quem é essa pessoa? Eu não a conheço. Vou mandá-la de volta para a tela de login". Isso tornaria o site impossível de usar.

Para resolver esse problema, precisamos de um jeito de fazer o servidor "lembrar" do usuário enquanto ele navega pelo site. Precisamos de um crachá mágico.

É aqui que entram as Sessões (em inglês, *Sessions*).

Pense em uma sessão como um crachá de identificação que você recebe ao entrar em um grande evento. Ao chegar, você mostra sua identidade (faz o login) uma única vez. Em troca, recebe um crachá com seu nome e nível de acesso.

A partir desse momento, você não precisa mais mostrar sua identidade em cada sala que entra. Basta exibir o crachá, e os seguranças (o nosso código PHP) sabem quem você é e onde tem permissão para ir. Essa é exatamente a função de uma sessão: ela é o crachá digital que identifica o usuário em todas as páginas do nosso site após o login.

2.2- Entregando o Crachá Após o Login.

2.2.1 - Criando a Sessão Segura

O momento de entregar o "crachá mágico" para o nosso usuário é imediatamente após a porta do castelo se abrir — ou seja, logo depois que a função password_verify() nos diz que a senha está correta.

Para fazer isso em PHP, usamos uma "supervariável" especial chamada $_SESSION. Pense nela como uma etiqueta em branco no crachá, onde podemos escrever as informações que queremos que o site lembre sobre o usuário, como seu ID único do banco de dados ou seu nome.

O processo tem duas etapas simples:

1. Avisar o PHP que vamos usar sessões: Antes de qualquer outra coisa no nosso código, precisamos colocar a função session_start();. Ela age como um aviso: "Ei, PHP, prepare o sistema de crachás, pois vamos precisar dele nesta página!".
2. Gravar os dados no crachá: Após o login ser validado, simplesmente guardamos as informações que queremos dentro da variável $_SESSION.

Vamos pegar nosso código de login da página 7 e completá-lo com a criação da sessão:

PHP

```
<?php

PASSO 0: INICIAR O SISTEMA DE SESSÕES
Esta função DEVE ser a primeira coisa no seu arquivo, antes de
qualquer HTML ou echo.
session_start();

 PASSO 1: RECEBER OS DADOS DO FORMULÁRIO DE LOGIN
$email_digitado = $_POST['email'];
$senha_digitada = $_POST['senha'];

PASSO 2: BUSCAR O HASH E OS DADOS DO USUÁRIO NO BANCO
(Aqui você faz a sua busca SQL pelo e-mail digitado)
```

```
Ex: SELECT id, nome, senha_hash FROM usuarios WHERE email =
'$email_digitado';

Vamos imaginar que os dados retornados do banco foram:

$id_do_usuario = 15;
$nome_do_usuario = "Maria Silva";
$hash_do_banco = '$2y$10$exemploDeHashSalvoNoSeuBancoDeDados';

 PASSO 3: VERIFICAR A SENHA E CRIAR O CRACHÁ
if (password_verify($senha_digitada, $hash_do_banco)) {

    Senha CORRETA! Hora de entregar o crachá.

    Gravamos o ID e o nome do usuário na sessão (no "crachá").

 $_SESSION['usuario_id'] = $id_do_usuario;
    $_SESSION['usuario_nome'] = $nome_do_usuario;

     Redirecionamos o usuário para a página principal do sistema.
    header('Location: painel_principal.php');
    exit();  //  É  uma  boa  prática  usar  exit()  após  um
redirecionamento.

} else {

    Senha INCORRETA.
    echo "E-mail ou senha inválidos.";
}
?>
```

Pronto! A partir de agora, enquanto o usuário navegar pelo site, o PHP saberá que o usuário de ID 15, chamado "Maria Silva", é quem está acessando as páginas. Mas como verificamos essa informação? É o que veremos a seguir.

2.3- Verificando o Crachá em Cada Sala.

2.3.1 - O "Segurança" de Cada Página

Agora que nosso usuário já tem seu crachá ($_SESSION), precisamos posicionar um "segurança" na porta de cada página que seja restrita, como o "Painel do Usuário", "Meus Agendamentos" ou a "Área de Configurações".

A função desse segurança é simples:

1. Verificar se a pessoa que está tentando entrar na página possui um crachá.
2. Se a pessoa tiver o crachá, a entrada é liberada.
3. Se a pessoa não tiver o crachá (ou seja, não fez login), ela é imediatamente acompanhada de volta para a portaria (a página de login).

Criar esse script de verificação é a forma mais inteligente de proteger nosso sistema. Em vez de escrevermos o mesmo código de checagem em todas as páginas, nós o criamos uma única vez em um arquivo separado e depois apenas "chamamos" o segurança quando precisamos dele.

Vamos criar um arquivo chamado verifica_login.php. Ele conterá nossa segurança:

PHP

```
<?php
verifica_login.php

PASSO 1: INICIAR O SISTEMA DE SESSÕES
Precisamos iniciar a sessão para conseguir ler as informações do crachá.
session_start();

 PASSO 2: VERIFICAR SE O CRACHÁ EXISTE
A função isset() verifica se uma variável existe.
Aqui, checamos se a informação 'usuario_id' NÃO FOI gravada no crachá.
O "!" no início significa "não".
if (!isset($_SESSION['usuario_id'])) {

    Se o crachá não existe, o usuário não fez login.
```

```
    Então, o mandamos de volta para a página de login.
    header('Location: login.php');
    exit(); // Encerra o script para garantir que nada mais seja executado.

}

Se o script chegou até aqui, significa que o crachá existe e é válido.
O segurança libera a passagem e a página restrita pode ser carregada.
?>
```

Agora, como usamos nosso segurança? É muito fácil. No topo de qualquer página que precise de proteção, como agenda.php, basta incluir este arquivo:

PHP

```
<?php
No topo do arquivo agenda.php
Chamamos nosso segurança para a porta desta página.
require_once 'verifica_login.php';
Se o código chegou até esta linha, o usuário está autenticado.
Todo o conteúdo da página de agendamentos vem aqui.
echo "<h1>Meus Agendamentos</h1>";
echo "Olá, " . htmlspecialchars($_SESSION['usuario_nome']). "!
Bem-vindo(a) à sua agenda.";
?>
```

Com essa técnica, proteger qualquer nova página se torna uma tarefa de adicionar apenas uma linha de código. É simples, eficiente e muito seguro.

2.4- Saindo do Evento: O Logout Seguro.

2.4.1 - Devolvendo o Crachá e Encerrando a Visita

Nosso usuário já conseguiu entrar no sistema, navegar pelas páginas restritas e agora quer sair. Um erro comum é pensar que simplesmente fechar a aba do navegador é o suficiente para encerrar a sessão. Isso seria como sair do

evento e jogar o seu crachá no lixo perto da porta. Alguém mal-intencionado poderia pegá-lo, colocá-lo e ter acesso às mesmas áreas que você tinha.

O processo de logout seguro garante que o crachá seja invalidado e completamente destruído, impossibilitando seu uso futuro. Em PHP, fazemos isso com um pequeno script, que geralmente fica em um arquivo chamado logout.php.

A lógica é a seguinte:

1. Iniciamos o sistema de sessões para ter acesso ao crachá atual.
2. Limpamos todas as informações gravadas no crachá (session_unset).
3. Destruímos o crachá em si, invalidando-o completamente (session_destroy).
4. Acompanhamos o usuário de volta para a portaria (página de login).

Veja como o código para isso é simples e direto:

PHP

```
<?php
// logout.php

// --- PASSO 1: INICIAR O SISTEMA DE SESSÕES ---
// Precisamos iniciar a sessão para poder manipulá-la.
session_start();

// --- PASSO 2: LIMPAR AS INFORMAÇÕES DO CRACHÁ ---
// A função session_unset() apaga todas as variáveis da sessão,
// como 'usuario_id' e 'usuario_nome'. O crachá fica em branco.
session_unset();

PASSO 3: DESTRUIR O CRACHÁ
A função session_destroy() invalida a sessão do usuário no servidor.
O crachá deixa de existir oficialmente.
session_destroy();
```

```
PASSO 4: REDIRECIONAR PARA A PÁGINA DE LOGIN
Após o logout, enviamos o usuário para um local seguro,
geralmente a tela de login.
header('Location: login.php');
exit();
?>
```

Com este script, garantimos um ciclo de vida completo e seguro para a autenticação do usuário. Ele entra, é identificado e, ao sair, todos os vestígios de sua sessão são completamente removidos, protegendo sua conta de acessos indevidos. Agora, estamos prontos para a próxima camada de defesa: proteger nossos formulários.

Capítulo 3: O Porteiro Atento: Filtrando Dados Maliciosos

3.1- SQL Injection- O Ataque do "Cavalo de Troia".

3.1.1 - Quando o Formulário Vira uma Arma

Até agora, focamos em proteger quem entra e navega em nosso sistema. Agora, precisamos nos defender do que as pessoas podem **enviar** para nós através dos formulários. Nosso sistema precisa de um "porteiro atento" que inspecione todos os pacotes de dados que chegam, antes de deixá-los entrar.
Um dos ataques mais clássicos e devastadores da internet é o **SQL Injection** (Injeção de SQL).

Para entendê-lo, imagine que seu banco de dados é um guardião de segredos muito obediente, mas que só entende uma língua específica: o SQL. Quando você quer uma informação, você faz um pedido a ele nessa língua. Por exemplo, em uma tela de login, seu código PHP faz o seguinte pedido:
"Guardião, por favor, encontre o usuário cujo e-mail é 'maria@email.com'."
O guardião entende perfeitamente e traz o registro da Maria.

O ataque de SQL Injection acontece quando um invasor, em vez de digitar um e-mail normal no campo do formulário, digita um pedaço da língua do guardião. Ele esconde um "Cavalo de Troia" dentro do texto. O pedido que chega ao guardião se transforma em algo assim:
"Guardião, por favor, encontre o usuário cujo e-mail é '' OU '1'='1'."

O que aconteceu aqui? O invasor fechou o campo de e-mail com aspas, adicionou um "OU" e uma condição que é **sempre verdadeira** ('1'='1'). Para o guardião obediente, a ordem agora é: "Encontre um usuário com e-mail em branco... ou, quer saber, apenas me traga **TODOS os usuários do banco**, porque 1 é sempre igual a 1!".

De repente, com uma única frase maliciosa, o invasor enganou o guardião e o fez entregar a lista completa de usuários e senhas. É por isso que o ataque

é tão perigoso. O formulário, que deveria ser apenas uma porta de entrada para dados, vira uma arma.

Felizmente, existe uma forma muito eficaz de ensinar nosso guardião a nunca ser enganado. Vamos ver como na próxima página.

3.2- A Defesa: Prepared Statements

3.2.1 - Conversando de Forma Segura com o Banco de Dados

Para evitar que nosso obediente "guardião" (o banco de dados) seja enganado, precisamos mudar a forma como conversamos com ele. Em vez de lhe dar uma ordem completa e complexa de uma só vez, vamos usar uma abordagem em duas etapas, muito mais segura, conhecida como Prepared Statements (em português, "Instruções Preparadas").

Funciona assim:

1. Primeiro, enviamos um "molde" da ordem: Nós entregamos ao guardião um papel com a ordem pré-aprovada, mas com lacunas em branco para os dados. Por exemplo:

"Guardião, sua tarefa é encontrar um usuário cujo e-mail é: [_____]. Entendido?"

Neste ponto, o guardião entende a intenção da tarefa, mas ainda não recebeu o dado. A estrutura do comando está travada e não pode ser alterada.

2. Depois, enviamos os dados separadamente: Em um segundo momento, entregamos ao guardião o que o usuário digitou no formulário, e dizemos: "Agora, preencha a lacuna da tarefa anterior com este valor aqui."

Com essa abordagem, o dado que o usuário envia é sempre tratado como... bem, apenas um dado. Ele nunca é interpretado como parte do comando. Mesmo que um invasor envie um "Cavalo de Troia", o guardião irá simplesmente procurar por um usuário com o nome bizarro " OU '1'='1', em vez de executar o comando. A armadilha é totalmente desarmada.

Em PHP, usando a extensão PDO (uma forma moderna de se conectar a bancos de dados), o código fica assim:

PHP

```
<?php

Supondo que a variável $pdo já contém sua conexão com o banco de dados.

PASSO 1: RECEBER O DADO DO FORMULÁRIO
$email_do_formulario = $_POST['email'];


PASSO 2: PREPARAR O "MOLDE" DA ORDEM
Note o ponto de interrogação (?). Ele é a lacuna que será preenchida.
$comando_preparado = $pdo->prepare("SELECT * FROM usuarios WHERE email = ?");


PASSO 3: EXECUTAR A ORDEM, ENVIANDO OS DADOS SEPARADAMENTE
A função execute() pega os dados do formulário e os encaixa de forma segura
na lacuna do comando que preparamos.
$comando_preparado->execute([$email_do_formulario]);

PASSO 4: PEGAR OS RESULTADOS
Agora podemos buscar os resultados da consulta com segurança.
Se um usuário for encontrado, ele estará na variável $usuario.
$usuario = $comando_preparado->fetch();

A partir daqui, você pode verificar se o $usuario foi encontrado e continuar o fluxo.
if ($usuario) {
```

```
    Usuário existe...
} else {
    Usuário não encontrado...
}
?>
```

Ao usar *Prepared Statements*, você cria uma barreira quase impenetrável contra ataques de SQL Injection. É a forma profissional e mais segura de interagir com seu banco de dados.

3.3- Cross-Site Scripting (XSS)- A "Pichação" Digital.

3.3.1 - Quando o Site Começa a Exibir Códigos Estranhos

Nosso "porteiro atento" já está inspecionando os pacotes que chegam para conversar com o banco de dados. Mas existe outro tipo de perigo: pacotes que contêm um código malicioso que não ataca o banco de dados, mas sim os **outros usuários** que visitam nosso site.

Esse ataque é conhecido como Cross-Site Scripting (XSS).

A melhor forma de entendê-lo é imaginá-lo como uma "pichação" digital.

Pense em uma área do seu site onde um usuário pode escrever algo que ficará visível para os outros, como uma seção de comentários, um perfil de usuário ou um post em um blog. Agora, imagine que um invasor, em vez de escrever um texto normal, "picha" na parede um código de script disfarçado de comentário.

Por exemplo:

<script> // Código malicioso aqui </script>

Quando outro usuário, uma vítima inocente, visita aquela página para ler os comentários, o navegador dela não vê aquilo como um simples texto. O navegador é projetado para ler e executar scripts, então ele pensa que aquela "pichação" é uma ordem legítima do seu site e a executa sem questionar.

O que esse código malicioso pode fazer?

- Roubar o "crachá mágico" (a sessão) do usuário e enviá-lo para o invasor, permitindo que ele se passe pela vítima.
- Redirecionar o usuário para um site falso para roubar mais informações.
- Exibir anúncios falsos ou alterar o conteúdo da sua página.

O perigo do XSS é que ele usa a confiança que o usuário tem no **seu site** para atacá-lo. A pichação foi feita na sua parede, então o navegador do visitante acredita que ela é parte da decoração original.

Felizmente, a defesa contra esse tipo de pichação é bastante simples. Precisamos apenas de uma "camada de verniz" para neutralizar qualquer código antes de exibi-lo. Vamos ver como fazer isso.

3.4- A Defesa: Limpando o que Será Exibido.

3.4.1 - Colocando um Vidro na Frente da Pichação

A solução para impedir a "pichação digital" (XSS) é surpreendentemente simples. Não precisamos tentar adivinhar se um texto é malicioso ou não. Em vez disso, adotamos uma regra de ouro: tratar todo e qualquer texto que venha de um usuário como se fosse potencialmente perigoso e "neutralizá-lo" antes de exibi-lo na tela.

Pense nisso como colocar uma **folha de vidro transparente** na frente da parede onde a pichação foi feita. Você ainda consegue ler o que está escrito, mas não pode mais tocar na tinta. A pichação se torna uma imagem inofensiva, em vez de uma substância ativa.

Em PHP, a nossa "folha de vidro" é uma função chamada htmlspecialchars().

O que essa função faz? Ela pega um texto e converte os caracteres que são especiais para o HTML (como <, >, &, ") em seus equivalentes de texto simples. Por exemplo, o caractere < vira <. Para o olho humano, a aparência é a mesma, mas para o navegador, a mágica acontece: ele passa a ler aquilo como um simples texto a ser exibido, e não como o início de uma tag de script a ser executada.

A regra é: sempre que você for exibir no HTML um dado que veio do banco de dados e foi inserido por um usuário, passe-o pela função htmlspecialchars().

Veja a diferença na prática:

PHP

```
<?php

DADO MALICIOSO
Imagine que um invasor salvou este "comentário" no seu banco de dados.
$comentario = '<script>alert("Você foi hackeado!");</script>';


MANEIRA INCORRETA E VULNERÁVEL DE EXIBIR
Se você fizer isto, o navegador executará o script e um alerta aparecerá.
echo "<div>" . $comentario . "</div>";


echo "<hr>";            // Linha para separar os exemplos


 MANEIRA CORRETA E SEGURA DE EXIBIR
Passamos o dado por uma "folha de vidro" (escape) antes de mostrar.
echo "<div>" . htmlspecialchars ($comentario, ENT_QUOTES, 'UTF-8') . "</div>";
O resultado será o texto literal "<script>alert(...)</script>" exibido na tela,
sem executar nada — o ataque fica neutralizado.
?>
```

Adotar o hábito de usar htmlspecialchars() em todas as saídas de dados é como ter um porteiro que não só inspeciona o que entra, mas também garante que tudo o que sai para ser exibido ao público esteja limpo e seguro. Com essa camada de proteção, fechamos as principais portas para os ataques mais comuns da web.

Capítulo 4: O Manual do Respeito: A LGPD na Prática

4.1- LGPD- O Que o Desenvolvedor Precisa Saber?

4.1.1 - Mais que uma Lei, um Compromisso com a Privacidade

Construímos um sistema que protege o acesso e se defende de ataques. Agora, precisamos falar sobre a camada mais importante de proteção: o respeito aos dados do usuário. No Brasil, esse respeito é guiado por uma lei chamada LGPD (Lei Geral de Proteção de Dados).

Para nós, desenvolvedores, a LGPD não deve ser vista como um obstáculo burocrático, mas como um manual de boas práticas que formaliza algo que já deveria ser nossa prioridade: a privacidade e a confiança de quem usa nosso sistema. Em um projeto de impacto social, isso é inegociável.

Não precisamos ser advogados para entender o essencial. A lei é vasta, mas para o nosso dia a dia de programação, podemos resumi-la em três compromissos fundamentais:

1. Consentimento: "Pergunte antes de usar."

Isso significa que o usuário deve concordar de forma clara e ativa com o uso de seus dados. Na prática, isso se traduz em ter, no formulário de cadastro, uma caixa de seleção (checkbox) com a frase "Li e aceito os Termos de Uso e a Política de Privacidade". Essa caixa não pode vir pré-marcada. O usuário precisa fazer a ação de clicar.

2. Finalidade: "Seja claro sobre o porquê de pedir um dado."

Só devemos coletar os dados que são estritamente necessários para o funcionamento do nosso serviço. Se pedimos o CEP do usuário, por exemplo, devemos informar que é para "encontrar profissionais próximos a ele". Não devemos pedir o CPF se não há uma razão clara e legal para isso. Transparência é a chave.

3. Direito ao Esquecimento: "O usuário é o dono dos dados e pode pedir para apagá-los."

Este é um dos direitos mais importantes. A plataforma é apenas uma "guardiã" temporária dos dados; o verdadeiro dono é o usuário. Portanto, nosso sistema obrigatoriamente precisa oferecer uma forma fácil e clara para que o usuário possa excluir sua conta e todos os seus dados de forma permanente.

É neste terceiro compromisso que nosso código tem a maior responsabilidade. Vamos ver a seguir como planejar e construir essa funcionalidade essencial.

4.3- O Direito ao Esquecimento- O Fluxo da Exclusão.

4.3.1 - Construindo a Porta de Saída

Oferecer o "direito ao esquecimento" significa que precisamos construir uma "porta de saída" clara, segura e definitiva em nosso sistema. Essa funcionalidade não pode estar escondida em menus complicados; ela deve ser acessível e seu funcionamento, transparente.

Antes de escrever o código, vamos desenhar a jornada do usuário ao decidir excluir sua conta. Um fluxo bem planejado garante segurança e evita exclusões acidentais.

Passo 1: O Ponto de Partida (Onde o botão mora?)

O link ou botão para iniciar o processo de exclusão deve estar em um local lógico, geralmente na área de "Configurações da Conta" ou "Meu Perfil". O texto deve ser inequívoco, como "Excluir minha conta".

Passo 2: A Confirmação de Segurança (Quem está pedindo?)

Clicar no botão não deve apagar tudo imediatamente. Isso seria perigoso. Em vez disso, o usuário deve ser levado a uma página de confirmação. Nesta página, para garantir que é realmente o dono da conta que está fazendo o pedido, a prática mais segura é solicitar que ele digite sua senha atual novamente. Isso previne que alguém mal-intencionado, que encontre uma sessão logada em um computador público, por exemplo, consiga apagar a conta de outra pessoa.

Passo 3: O Último Aviso (Você tem certeza absoluta?)

Junto ao campo de senha, devemos exibir um aviso final, em destaque e com linguagem forte e clara. Algo como:

Atenção! Esta ação é permanente e não pode ser desfeita.

Ao confirmar, todos os seus dados, incluindo seu perfil, histórico de agendamentos e informações salvas, serão apagados para sempre de nossa plataforma.

Passo 4: A Execução e a Mensagem de Despedida

Somente após o usuário digitar a senha correta e clicar no botão final ("Confirmar exclusão permanente"), nosso código no servidor entrará em ação para remover os dados. Uma vez que o processo termine, o usuário deve ser deslogado automaticamente e redirecionado para a página inicial do site com uma mensagem de confirmação, como "Sua conta foi excluída com sucesso".

Com este fluxo, garantimos que a exclusão seja uma decisão consciente, verificada e segura, cumprindo nossa obrigação com a LGPD e respeitando o usuário até o último momento.

4.4- Mão na Massa- O Código por Trás do "Apagar Tudo".

4.4.1 - Executando a Exclusão no Banco de Dados

Com o fluxo de exclusão bem definido, é hora de programar a ação final. O código a seguir representa o que aconteceria no servidor após o usuário confirmar sua senha na página de exclusão.

Este script tem uma grande responsabilidade, então ele segue uma lógica de verificação rigorosa antes de apagar qualquer coisa.

Ponto crucial: Para cumprir a LGPD, não basta apagar o registro do usuário na tabela principal. Precisamos apagar todos os dados associados a ele em qualquer outra tabela do sistema (agendamentos, posts de blog, comentários, etc.).

A forma mais segura de fazer isso é apagar primeiro os "dados filhos" para depois apagar o "registro pai".

Veja um exemplo de como seria o código processa_exclusao.php:

PHP

```
<?php
 processa_exclusao.php

session_start();

Garante que o usuário esteja logado antes de tentar excluir a
conta.
if (!isset($_SESSION['usuario_id'])) {
    header('Location: login.php');
    exit();
}

   1.  Aqui assumimos que a variável $pdo já tem a conexão com o
      banco.
      Pega o id do usuário que está salvo na sessão.
      $id_do_usuario_logado = $_SESSION['usuario_id'];

      Pega a senha que o usuário digitou no formulário de
confirmação.
      $senha_digitada = $_POST['senha'];

   2. Buscar o hash da senha do usuário no banco.
      $sql_busca = $pdo->prepare(“SELECT senha_hash FROM usuarios
           WHERE id = ?”);
      $sql_busca->execute([$id_do_usuario_logado]);
      $dados_do_usuario = $sql_busca->fetch();

      3. Verificar se a senha digitada confere com o hash do
banco.
```

```
if ($dados_do_usuario && password_verify($senha_digitada,
    $dados_do_usuario['senha_hash'])) {

SENHA CORRETA — seguir com a exclusão.

4. Apagar dados relacionados (tabelas filhas).

Exemplo: apagar todos os agendamentos desse usuário.
$sql_apaga_agendamentos = $pdo->prepare("DELETE FROM
agendamentos WHERE id_usuario = ?");
$sql_apaga_agendamentos->execute([$id_do_usuario_logado]);

(Adicione aqui outros DELETE para outras tabelas
relacionadas, se houver.)

5. Apagar o registro principal do usuário (tabela pai).

$sql_apaga_usuario = $pdo->prepare("DELETE FROM usuarios
WHERE id = ?");
 $sql_apaga_usuario->execute([$id_do_usuario_logado]);

6. Limpar e destruir a sessão (logout).
session_unset();
session_destroy();


7. Redirecionar para a página inicial com uma mensagem de
sucesso.

 header('Location: index.php?mensagem=conta_excluida');
 exit();

} else {
```

```
        // SENHA INCORRETA.
       // Voltar para a página de exclusão com um erro.
        header('Location:
pagina_de_exclusao.php?erro=senha_invalida');
        exit();
       }
       ?>
```

Com este código, garantimos que a exclusão só aconteça sob as condições certas, seguindo uma ordem lógica e segura, e que ao final do processo, o usuário seja completamente desvinculado da plataforma, cumprindo de forma prática e definitiva o seu direito ao esquecimento.

Próximos Passos: Da Teoria à prática continua

Checklist Final de Segurança

Sua Lista de Verificação Antes de Lançar um Projeto

Passamos pela teoria e pela prática, construindo camada por camada a segurança do nosso projeto. Mas o conhecimento só se torna poderoso quando o transformamos em ação consistente.

Para te ajudar a aplicar tudo o que aprendemos, criamos este checklist final. Antes de colocar qualquer projeto seu no ar, passe por cada um destes itens. Pense nele como a inspeção final de um piloto antes de decolar. Não pule nenhuma etapa. A segurança do seu projeto e a confiança dos seus usuários dependem disso.

[ ] Proteção de Senhas (Capítulo 1)

- Eu estou usando a função password_hash() para salvar TODAS as senhas de usuários no banco de dados?
- Eu estou usando a função password_verify() para comparar as senhas durante o login?
- Eu NUNCA salvo a senha original do usuário em nenhum lugar?

[ ] Controle de Acesso com Sessões (Capítulo 2)

- Todas as minhas páginas restritas incluem um script no topo que verifica se o usuário está logado (!isset($_SESSION['usuario_id']))?
- O meu script de logout.php limpa (session_unset()) e destrói (session_destroy()) a sessão completamente?

[ ] Defesa Contra-ataques (Capítulo 3)

- TODAS as minhas consultas ao banco de dados que usam dados vindos de um usuário (formulários, links, etc.) estão usando **Prepared Statements** (com PDO ou MySQLi)?

- Eu estou usando a função htmlspecialchars() para tratar TODAS as informações que vêm de um usuário antes de exibi-las em uma página HTML?

[ ] Conformidade com a LGPD (Capítulo 4)

- Meu sistema oferece uma forma clara e acessível para o usuário excluir sua própria conta?
- O meu script de exclusão apaga TODOS os dados associados àquele usuário, não apenas o registro na tabela principal?
- A exclusão de conta exige uma confirmação de segurança (como digitar a senha atual) para evitar acidentes?

Use esta lista como seu guia. Se você puder marcar "sim" para todas essas perguntas, você estará muito à frente na construção de uma aplicação robusta, segura e digna da confiança dos seus usuários.

Mantendo a Fortaleza Segura: Boas Práticas Contínuas.

Protegendo o Terreno: Segurança Fora do Código PHP

Nossa jornada até aqui focou em construir um código seguro. Mas uma casa forte, construída em um terreno instável, ainda corre o risco de desabar. Da mesma forma, nosso código seguro precisa rodar em um ambiente (servidor) igualmente protegido.

Existem duas práticas essenciais que cuidam da segurança fora do seu código e que são cruciais para a proteção do seu projeto.

1. O Cadeado no Endereço do Site (HTTPS)

Você já reparou no pequeno cadeado que aparece ao lado do endereço da maioria dos sites hoje em dia? Aquilo significa que o site usa **HTTPS**, e não apenas HTTP.

Pense nisso da seguinte forma:

- HTTP é como enviar uma carta por correio em um cartão-postal. Qualquer pessoa no caminho (provedores de internet, hackers em redes Wi-Fi públicas) pode ler o conteúdo. Se um usuário envia uma senha, ela viaja em texto puro.
- HTTPS é como enviar a mesma carta dentro de um envelope de aço, lacrado. A comunicação entre o navegador do usuário e o seu servidor é criptografada. Mesmo que alguém intercepte os dados, verá apenas um código embaralhado e sem sentido.

Por que é vital? Sem HTTPS, todas as senhas, e-mails e dados pessoais que seus usuários enviam podem ser facilmente roubados. Hoje, obter o "cadeado" (um certificado SSL/TLS) é fácil e, em muitos casos, gratuito, através de serviços como o Let's Encrypt, que já vêm integrados na maioria das hospedagens. Não há desculpa para não usá-lo.

2. Escondendo os Detalhes dos Erros em Produção

Quando estamos desenvolvendo, os erros detalhados que o PHP nos mostra são nossos melhores amigos. Eles nos dizem a linha exata do problema e o que deu errado.

No entanto, quando o site está no ar (em "produção"), esses mesmos erros se tornam nossos piores inimigos. Eles agem como um funcionário demitido e descontente que grita os segredos da empresa para qualquer um que queira ouvir. Um erro detalhado pode revelar:

- O caminho completo dos seus arquivos no servidor.
- Nomes de tabelas e colunas do seu banco de dados.
- As credenciais de conexão com o banco (em casos graves de má configuração).

A Prática Correta: No servidor de produção, configure o PHP para nunca exibir erros na tela. Em vez disso, ele deve gravá-los em um arquivo de log privado, que só você pode acessar.

Como fazer: No arquivo de configuração do PHP (php.ini), garanta que estas duas linhas estejam assim:

display_errors = Off

log_errors = On

Guardando o Mapa do Tesouro em um Lugar Secreto.

Até o porteiro mais atento e a fortaleza mais segura se tornam inúteis se o invasor conseguir uma cópia da chave mestra. No nosso projeto, a "chave mestra" são as credenciais de acesso ao banco de dados: o nome do servidor, o nome do banco, o usuário e, principalmente, a senha.

3. O Cofre das Credenciais: O Arquivo de Configuração

Um erro muito comum entre iniciantes é colocar as credenciais do banco de dados diretamente no código, em arquivos que são publicamente acessíveis pela web. Isso é o mesmo que anotar a senha do cofre em um post-it e colá-lo na porta do cofre.

A Prática Correta: As credenciais devem estar em um arquivo de configuração separado (ex: config.php) e, o mais importante, este arquivo deve ser armazenado fora da pasta pública do seu site.

Imagine que a estrutura de arquivos do seu servidor é assim:

```
/home/usuario/
├── logs/                 (Aqui ficam os logs de erro)
├── private/              (Pasta privada, não acessível pela web)
│   └── config.php        <-- O MAPA DO TESOURO FICA AQUI!
└── public_html/          (A pasta que o mundo vê)
    ├── index.php
    ├── login.php
    └── css/
```

No seu código `index.php`, você "chama" o arquivo de configuração de forma segura:

```
require_once '../private/config.php';
```

Dessa forma, mesmo que uma falha de configuração no servidor exponha o código dos seus arquivos PHP, o invasor nunca conseguirá ler o conteúdo do config.php, pois ele está em um local inacessível pela internet.

4. A Vacina Constante: A Importância das Atualizações

Nenhum software é perfeito. O próprio PHP, o servidor web que você usa (como Apache) e quaisquer outras ferramentas que seu projeto utilize são constantemente revisados por uma comunidade global de desenvolvedores que procuram por falhas de segurança.

Quando uma falha é descoberta, ela é corrigida e uma nova versão do software é lançada. Manter seu ambiente desatualizado é como se recusar a tomar uma vacina para uma doença recém-descoberta. Você fica exposto a vulnerabilidades que já são conhecidas e para as quais já existe uma cura.

A Prática Correta:

- **Use Versões Suportadas do PHP:** Sempre utilize uma versão do PHP que esteja recebendo ativamente atualizações de segurança. Versões muito antigas são como fortalezas abandonadas, cheias de buracos conhecidos.
- **Mantenha Tudo Atualizado:** Periodicamente, verifique e aplique as atualizações para o seu servidor, banco de dados e quaisquer bibliotecas ou frameworks que seu projeto utilize.

A segurança não é um evento único, mas um ciclo de vigilância. Proteger suas credenciais e manter seu ambiente atualizado são duas das práticas mais poderosas para garantir a longevidade e a integridade da sua fortaleza digital.

Conclusão

O Código Como um Ato de Cuidado

Ao chegar ao fim deste guia, você não aprendeu apenas "receitas" de código. Qualquer um pode copiar e colar uma função. O que você aprendeu foi algo muito mais importante: os princípios da **higiene digital** que separam um programador amador de um profissional responsável.

Pense nisso como um juramento silencioso que todo bom desenvolvedor faz. Assim como um médico jura proteger a vida de seus pacientes, ou um engenheiro jura construir pontes seguras, nós, ao lidarmos com dados, juramos proteger a privacidade e a confiança das pessoas que usam nossas criações.

Em um projeto de impacto social, esse juramento é ainda mais forte. As informações que guardamos não são apenas texto em uma tela; são as histórias, as necessidades e a vulnerabilidade de uma comunidade que confia em nosso trabalho para ter um ambiente seguro.

Cada linha de código de segurança que você aprendeu a escrever nestas páginas — desde proteger uma senha até criar uma porta de saída para os dados — não foi um mero detalhe técnico.

Foi um ato de respeito. Foi a sua forma de dizer para cada usuário: "Eu me importo com você e com a sua segurança".

A tecnologia é um rio que nunca para de correr. Novas ameaças e novas soluções surgirão amanhã. Mas o princípio fundamental que você aprendeu aqui nunca vai mudar: a base de qualquer projeto de sucesso é a confiança.

Portanto, leve este conhecimento não como um ponto final, mas como o verdadeiro começo da sua jornada como desenvolvedor.

Lembre-se sempre: **um bom código faz um site funcionar. Mas um código seguro e responsável faz um projeto ser digno de existir.**

Referências

Este guia foi construído com base nas melhores práticas recomendadas pela comunidade de desenvolvimento e segurança web. Para aprofundamento nos tópicos abordados, as seguintes fontes oficiais são altamente recomendadas.

Documentação Oficial do PHP

PHP. **Função password_hash()**. Disponível em: https://www.php.net/manual/pt_BR/function.password-hash.php. Acesso em: 21 out. 2025.

PHP. **Função password_verify()**. Disponível em: https://www.php.net/manual/pt_BR/function.password-verify.php. Acesso em: 21 out. 2025.

PHP. **Funções de Sessão**. Disponível em: https://www.php.net/manual/pt_BR/book.session.php. Acesso em: 21 out. 2025.

PHP. **PHP Data Objects (PDO)**. Disponível em: https://www.php.net/manual/pt_BR/book.pdo.php. Acesso em: 21 out. 2025.

PHP. **Função htmlspecialchars()**. Disponível em: https://www.php.net/manual/pt_BR/function.htmlspecialchars.php. Acesso em: 21 out. 2025.

Segurança Web e Boas Práticas

OWASP FOUNDATION. **OWASP Top 10**. A lista das dez vulnerabilidades mais críticas em aplicações web. Disponível em: https://owasp.org/www-project-top-ten/. Acesso em: 21 out. 2025.

OWASP FOUNDATION. **Injection Flaws**. Explicação detalhada sobre ataques de injeção. Disponível em: https://owasp.org/www-community/attacks/Injection_Flaws. Acesso em: 21 out. 2025.

OWASP FOUNDATION. **Cross Site Scripting (XSS)**. Explicação detalhada sobre ataques de XSS. Disponível em: https://owasp.org/www-community/attacks/xss/. Acesso em: 21 out. 2025.

MOZILLA. **HTTPS**. Artigo da MDN Web Docs explicando o que é e por que o HTTPS é importante. Disponível em: https://developer.mozilla.org/pt-BR/docs/Glossary/HTTPS. Acesso em: 21 out. 2025.

Legislação

BRASIL. **Lei nº 13.709, de 14 de agosto de 2018**. Lei Geral de Proteção de Dados Pessoais (LGPD). Disponível em: http://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm. Acesso em: 21 out. 2025.

Glossário de Termos

1 - Bcrypt

Um algoritmo (uma "receita matemática") moderno e muito seguro usado para criar hashes de senhas. É o método padrão utilizado pela função password_hash() do PHP por ser intencionalmente lento, o que o torna muito resistente a ataques de força bruta.

2 - Cross-Site Scripting (XSS)

Um tipo de ataque em que um invasor "picha" um código malicioso (um script) em uma página web. Quando outros usuários visitam essa página, o navegador deles executa o código, o que pode levar ao roubo de informações.

3 - Hash

O resultado de um processo de embaralhamento de dados que só funciona em uma direção. É como transformar uma senha em uma "vitamina" digital: é fácil criar a vitamina a partir da fruta, mas impossível transformar a vitamina de volta na fruta original.

4 - htmlspecialchars()

Uma função do PHP que age como uma "folha de vidro", neutralizando códigos maliciosos em um texto. Ela converte caracteres especiais de HTML em texto simples, garantindo que eles sejam exibidos de forma segura na tela sem serem executados pelo navegador.

5 - HTTPS (Hypertext Transfer Protocol Secure)

A versão segura do protocolo HTTP, usada para a comunicação entre o navegador e o servidor. O "S" significa que a conexão é criptografada (protegida por um "envelope de aço"), impedindo que invasores leiam os dados que são trocados, como senhas e informações pessoais

6 - LGPD (Lei Geral de Proteção de Dados)

A legislação brasileira que estabelece as regras sobre como as empresas e projetos devem coletar, armazenar e tratar os dados pessoais dos usuários. Seu principal objetivo é garantir a privacidade e dar às pessoas mais controle sobre suas próprias informações.

7 - MD5

Um algoritmo de hash antigo e hoje considerado inseguro para o armazenamento de senhas. Por ser muito rápido e vulnerável a colisões (diferentes dados que geram o mesmo hash), ele pode ser facilmente quebrado com técnicas modernas.

8 - OWASP (Open Web Application Security Project)

Uma fundação sem fins lucrativos e uma comunidade global focada em melhorar a segurança de software. A OWASP é famosa por publicar a "OWASP Top 10", uma lista das dez vulnerabilidades de segurança mais críticas em aplicações web, que serve como um guia para desenvolvedores.

9 - password_hash()

A função moderna do PHP recomendada para criar hashes de senhas. Ela automaticamente usa um algoritmo forte (como o Bcrypt) e adiciona um "salt" único a cada senha, tornando o processo extremamente seguro.

10 - password_verify()

A função do PHP "parceira" da password_hash(). É usada para verificar se uma senha digitada pelo usuário corresponde a um hash que foi salvo no banco de dados, sem nunca precisar reverter o hash. 8

11- PDO (PHP Data Objects)

Uma extensão do PHP que oferece uma interface moderna, segura e consistente para se conectar e interagir com bancos de dados. Seu principal recurso de segurança é o suporte a Prepared Statements.

12 - Prepared Statements (Instruções Preparadas)

A forma mais segura de executar comandos SQL. A técnica consiste em separar a "ordem" do "dado", enviando-os em duas etapas para o banco de dados. Isso cria uma barreira que impede ataques de SQL Injection.

13 - Salt

Um valor aleatório que é adicionado a uma senha antes de ela ser transformada em um hash. O "salt" garante que, mesmo que dois usuários tenham a mesma senha, o hash final guardado no banco de dados seja completamente diferente para cada um, tornando ataques de "catálogo" (como rainbow tables) inúteis.

14 - Sessão (Session)

Um mecanismo que permite que um site "lembre" de um usuário enquanto ele navega por diferentes páginas. Funciona como um "crachá mágico" que o usuário recebe após o login e que o identifica em todas as áreas restritas do site.

15 - SQL Injection (Injeção de SQL)

Um tipo de ataque em que um invasor insere comandos de banco de dados (SQL) maliciosos em um campo de formulário. Se o sistema não estiver protegido, ele pode ser enganado e executar esses comandos, o que pode levar ao vazamento ou à destruição de todos os dados.

16 - SSL/TLS (Secure Sockets Layer / Transport Layer Security)

O certificado digital que possibilita a comunicação segura via HTTPS. Ele é o responsável por criptografar os dados entre o navegador do usuário e o servidor, garantindo a confidencialidade e a integridade das informações. É o que "liga" o cadeado no navegador.