REVELAÇÕES PERIGOSAS NO CÓDIGO

A VERDADE POR TRÁS DO LAÇO: QUANDO O BACKEND FALA DEMAIS

Luiz Vitorio Casagrande

LAÇOS QUE REVELAM DEMAIS

🕸️ Quando a Resposta POST Entrega o Segredo

Falhas de exposição de dados podem ocorrer quando a resposta de uma requisição POST (como recuperação de senha ou envio de token) traz dados sensíveis como tokens diretamente no corpo da resposta. Esse comportamento é comum em APIs mal configuradas, onde o frontend recebe informações que deveriam ser processadas e armazenadas exclusivamente no backend.

Exemplo prático

Supondo um endpoint de login que responde com dados do perfil completo, incluindo permissões:

```json
{
  "username": "usuario1",
  "role": "admin",
  "canDeleteUsers": true
}
```

Mesmo que a interface não permita excluir usuários, um invasor pode simular essa ação com base na resposta. É inseguro pois expõe funções e permissões que deveriam ser verificadas e controladas no servidor. Pode permitir elevação de privilégio, se combinado com outras falhas.

Sugestão de Correção

A resposta do servidor deve conter apenas as informações necessárias para o cliente renderizar a interface ou confirmar ações. Permissões e verificações de segurança devem ser feitas somente no backend, jamais confiando no que vem do cliente.

```json
{
  "message": "Login realizado com sucesso"
}
```

📜 Profecias em JavaScript: O Código que Previu sua Própria Queda

JavaScript é executado no cliente e, portanto, todo seu conteúdo pode ser lido, analisado e explorado. Se esse código contém tokens, URLs internas, nomes de funções críticas ou dados sensíveis, qualquer pessoa pode acessar essas informações.

Exemplo prático

```js
viewApi = function(source){
   $.post("view_api_data.php", {
      token: "3c469e9d6c5875d37a43f353d4f88e61fcf812c66eee3457465a40b0da4153e0",
      source: source
   }, function(data){
      $('#content_api').html(data);
   });
};
```

Esse código expõe diretamente o token da API. Um invasor pode usá-lo para consultar dados internos apenas manipulando o parâmetro source. Eventuais riscos que podem ser levantados são a divulgação de tokens de API, injeção de parâmetros para consultar dados sensíveis, possível bypass de autenticação ou autorização.

Sugestão de Correção

É recomendado remover tokens do frontend. A lógica deve ser realizada no backend, como exemplo pode passar o filtro por GET e processar com controle interno. É também recomendável implementar whitelist de filtros permitidos.

💬 Sussurros do Código: Comentários HTML que Falam Demais

Durante o desenvolvimento, é comum deixar comentários no HTML. Porém, quando esses comentários chegam à produção, qualquer visitante pode inspecionar e ver essas anotações, que podem conter informações estruturais, rotas internas, campos sensíveis ou até dados de teste.

Exemplo prático

```html
<!— TODO: implementar autenticação mais forte —>
<!— acesso direto para debug: /admin/areaInterna.php —>
<!— credenciais: admin / 1234 —>

<!—
<div class="form-group">
   <label for="email">Password*</label>
   <input type="password" name="password" value="<?php echo $password; ?>">
</div>
—>
```

Comentários deixados no código e que podem ser lidos examinando o HTML podem expor nomes de campos sensíveis, indicar funcionalidades não visíveis, isso pode ser base para engenharia reversa.

Sugestão de Correção

Comentários visíveis no HTML devem ser removidos dos ambientes de produção, de preferência antes de sua publicação. Usar ambientes de staging ou revisão automatizada para garantir que esses comentários não cheguem à produção.

🤖 A Trilha dos Bots: Caminhos Secretos que Nunca Deveriam Ser Listados

O arquivo robots.txt é utilizado para instruir motores de busca a não indexarem certos caminhos da aplicação. Contudo, ele também revela esses caminhos a qualquer invasor que acesse o arquivo diretamente.

Exemplo prático

robots.txt

```
User-agent: *
Disallow: /admin/
Disallow: /backup/
```

O arquivo robots.txt traz alguns riscos como possíveis áreas sem autenticação, pode facilitar ataques de enumeração e fuzzing, bem como revelar diretórios internos.

Sugestão de Correção

Não listar diretórios sensíveis e usar mecanismos de controle de acesso (como autenticação, bloqueio por firewall) para protegê-los. Além disso, para evitar exposição, pode-se usar X-Robots-Tag.

php

```php
header('X-Robots-Tag: noindex, nofollow');
```

Outra possível solução seria utilizar meta tags em HTML sem revelar paths internos.

HTML

```html
<meta name="robots" content="noindex, nofollow">
```

Uma outra opção indicada seria configurar .htaccess para bloquear acesso a determinados tipos de arquivos

.htaccess

```
<Files ~ "\.xml$">
  Header set X-Robots-Tag "noindex, nofollow"
</Files>
```

02

PROTEGENDO OS SEGREDOS DA SUA THEMYSCIRA DIGITAL

🔒 Protegendo Senhas Como as Amazonas

Senhas armazenadas em texto plano ou com hash fraco facilitam vazamentos e ataques de força bruta. Um banco de dados comprometido pode expor as credenciais de todos os usuários.

Exemplo prático

```sql
INSERT INTO clients (email, password) VALUES ('email@exemplo.com', 'senha123')
```

```php
$senhaCriptografada = openssl_encrypt('senha123', 'aes256', $chave);
```

Alguns dos riscos advindos são o comprometimento total em caso de invasão, a reutilização de senhas expostas em outros sistemas levando a seu comprometimento, por fim cabe mencionar que a falta de salt permite uso de rainbow tables, ou seja, uma tabela pré-computada para reverter funções hash criptográficas.

Sugestão de Correção

Utilização de algoritmos robustos de hashing lento como bcrypt, scrypt ou Argon2, que já incluem o salt no resultado e são resistentes a ataques de força bruta.

🔐 Convertendo Senhas Antigas em Armas Novas

Em aplicações já em produção, é necessário atualizar senhas existentes sem impactar o login dos usuários.

Exemplo prático

```php
while ($usuario = $conn→fetch_array($query)){
    $salt = hash('sha1', $functions→createPassword());
    $newPassword = hash('sha256', md5($salt . hash('sha256', $usuario["password"])));
    // Atualiza banco
}
```

Se a criptografia for reversível, e a chave for roubada ou descoberta, o invasor pode acessar as senhas.

Sugestão de Correção

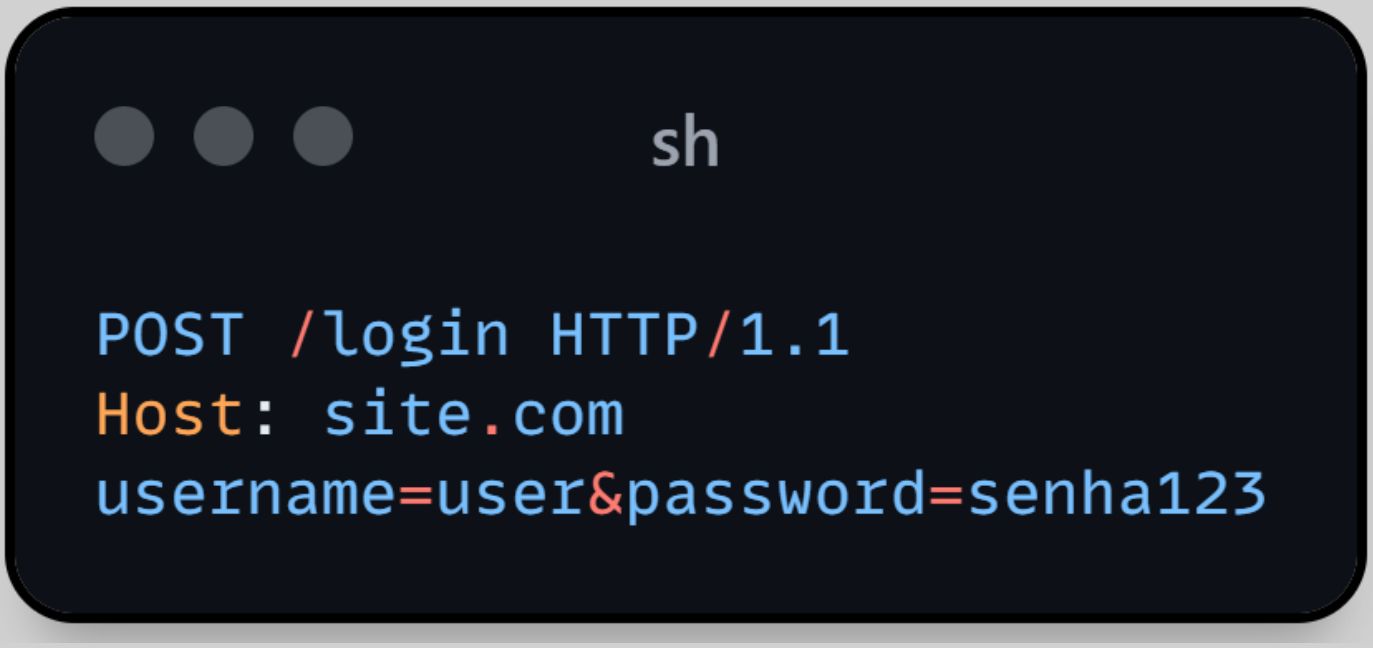
Executar com cuidado um script de migração, após ter validado em ambiente de testes. Se caso for necessário, ao próximo login do usuário, fazer novo hash com algoritmo moderno e descartar o hash antigo.

COMO O ATAQUE MAN-IN-THE-MIDDLE ROUBA SEM SER VISTO

⚔️ Como o Ataque Man-in-the-Middle Rouba sem Ser Visto

Ataques MITM ocorrem quando um invasor intercepta a comunicação entre o cliente e o servidor — especialmente quando é feita via HTTP, sem criptografia, pois, os dados trafegam em texto aberto e podem ser capturados facilmente, inclusive senhas e tokens.

Exemplo prático



Em redes abertas, isso pode ser capturado com ferramentas como Wireshark, toda a comunicação pode ser lida e alterada. Tokens de sessão podem ser roubados e reutilizados.

Sugestão de Correção

- Utilizar HTTPS sempre com certificado válido e redirecionamento forçado, vale destacar como defesa adicional o uso de certificate pinning;
- Utilizar HSTS para reforçar conexões seguras;
- Atualizar bibliotecas e evite SSL/TLS desatualizados (ex: SSLv3, TLS 1.0), utilize TLS 1.2 e 1.3.

🛡️ O Escudo Invisível: Nada deve trafegar desprotegido no campo de batalha

Mesmo que sua aplicação esteja disponível via HTTPS, um invasor pode tentar forçar conexões HTTP não seguras. Para mitigar isso, você deve redirecionar todo o tráfego HTTP automaticamente para HTTPS, garantindo que o navegador nunca faça uma conexão insegura.

Sugestão de Correção

- Habilitar HSTS para reforçar conexões seguras, nas configurações de header da página.
- Caso utilize Apache, pode ser feita a configuração semelhante no arquivo .htaccess:

```apache
RewriteEngine On
RewriteCond %{HTTPS} off
RewriteRule ^(.*)$ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

- Caso utilize Nginx, pode ser semelhante:

```nginx
server {
    listen 80;
    return 301 https://$host$request_uri;
}
```

🏁 Conclusão: A jornada da Amazona de Themyscira

A exposição de dados sensíveis é, muitas vezes, o resultado de pequenos descuidos acumulados no ciclo de desenvolvimento. Um comentário esquecido no HTML, um token hardcoded em um arquivo JavaScript, um diretório revelado no robots.txt ou uma senha armazenada sem a devida proteção podem parecer detalhes irrelevantes durante a programação — mas, para um invasor, são pistas valiosas que abrem caminho para uma invasão completa.

Na prática, proteger dados sensíveis envolve três pilares:

1. Minimizar a exposição: só entregar ao cliente o que ele realmente precisa.

2. Fortalecer a proteção: armazenar informações com algoritmos robustos, hashes e salting.

3. Blindar a comunicação: garantir criptografia de ponta a ponta, evitando interceptações.

A segurança de uma aplicação não está apenas nas grandes defesas, mas na soma das pequenas boas práticas. No desenvolvimento seguro, escolher fazer a coisa certa — mesmo nos detalhes — é o que garante que a verdade (os dados) permaneçam protegidos.


SENSITIVE
DATA
PROTECTED