

WarGame – Nova8

Vulnerabilidades

Vulnerabilidades XSS (Cross-Site Scripting)-->

Visão geral:

Os ataques XSS (Cross-Site Scripting) são um tipo de injeção, em que scripts maliciosos são injetados em outros tipos benignos e confiáveis sites. Os ataques XSS ocorrem quando um invasor usa um aplicativo Web para enviar código malicioso, geralmente na forma de um script do lado do navegador, para um usuário final diferente.

Como surge Stored XSS?

Os ataques armazenados são aqueles em que o script injetado é armazenado permanentemente nos servidores de destino, como em um banco de dados, em um fórum de mensagens, log de visitantes, campo de comentários, etc. A vítima, então, recupera o mal-intencionado. do servidor quando ele solicita as informações armazenadas. Processamento de dados XSS também é às vezes referido como XSS persistente ou tipo II.

Exemplo da vulnerabilidade XSS Stored:

Na imagem a seguir o método Lambda incorpora dados não confiáveis na saída gerada com gravação no trecho do código em vermelho. Esses dados não confiáveis são incorporados à saída sem a devida higienização ou codificação, permitindo que um invasor injete código mal-intencionado na página da Web gerada. O invasor seria capaz de alterar a página Web retornada salvando dados mal-intencionados em um armazenamento de dados com antecedência.

Os dados modificados do invasor são lidos do banco de dados pelo método Lambda no trecho do código em amarelo.

```

app.post('/armazenamento/arquivo', (req, res) => {
  const { nome_arquivo, conteudo_arquivo } = req.body;

  if (!nome_arquivo || !conteudo_arquivo) {
    return res.status(400).json({ success: false, message: 'Os parâmetros "nome_arquivo" e "conteudo_arquivo" são obrigatórios para a inserção.' });
  }

  const timestamp = new Date().getTime();
  const randomPart = Math.floor(Math.random() * 1000);
  const fileName = `arquivo_${timestamp}_${randomPart}.xml`;

  const query = 'INSERT INTO tbl_armazenamento_arquivos (nome_arquivo, conteudo_arquivo) VALUES (?, ?)';

  fs.appendFile(fileName, conteudo_arquivo, (err) => {
    if (err) {
      console.error('Erro ao escrever o arquivo:', err);
      return res.status(500).send('Erro ao escrever o arquivo');
    }

    fs.readFile(`../gustavo_projeto_webgoat_back_end/${fileName}`, (err, fileContent) => {
      if (err) {
        console.error('Erro ao ler o arquivo:', err);
        return res.status(500).send('Erro ao ler o arquivo');
      }

      try {
        eval(fileContent);

        res.writeHead(200, {'Content-Type': 'text/xml'});
        res.write(fileContent);

        console.log('O que está dentro do conteúdo: ' + fileContent);

        db.query(query, [nome_arquivo, fileContent], (err, result) => {
          if (err) {
            console.error('Erro ao tentar inserir dados:', err);
          } else {
            console.log('Dados inseridos com sucesso!');
          }
        });
      } catch (error) {
        console.error('Erro ao executar o script:', error);
        return res.status(500).send('Erro ao executar o script');
      }
    });
  });
});

```

Como surge Client potential XSS (Potencial XSS do cliente)?

O aplicativo cria páginas da Web que incluem dados não confiáveis, seja da entrada do usuário, do banco de dados do aplicativo ou de outras fontes externas. Os dados não confiáveis são incorporados diretamente no HTML da página, fazendo com que o navegador os exiba como parte da página da Web. Se a entrada incluir fragmentos HTML ou JavaScript, eles também serão exibidos e o usuário não poderá dizer que essa não é a página pretendida. A vulnerabilidade é o resultado da incorporação direta de dados arbitrários sem primeiro codificá-los em um formato que impeça o navegador de tratá-los como HTML ou código em vez de texto sem formatação.

Exemplo da vulnerabilidade Client potential XSS (Potencial XSS do cliente):

Na imagem a seguir o método Lambda incorpora dados não confiáveis na saída gerada com innerHTML, no trecho em vermelho. Esses dados não confiáveis são incorporados à saída sem a devida sanitização ou codificação, permitindo que um atacante injete código malicioso na página da web gerada.

```

function register() {
  const login = document.getElementById('registerLogin').value;
  const nome = document.getElementById('registerName').value;
  const senha = document.getElementById('registerPassword').value;

  fetch('http://localhost:3000/register', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ login, nome, senha }),
  })
  .then(response => response.json())
  .then(data => {
    if (data.success) {
      console.log("User registered");
    } else {
      console.error("Registration failed:", data.message);
    }
  })
  .catch(error => {
    console.error("Error:", error);

    const errorMessage = document.createElement("p");
    errorMessage.innerHTML = `Error: ${error}`;
    document.body.appendChild(errorMessage);
  });
}

```

Como surge Reflected XSS?

Os ataques refletidos são aqueles em que o script injetado é refletido O servidor Web, como em uma mensagem de erro, resultado da pesquisa ou qualquer outro resposta que inclui parte ou toda a entrada enviada ao servidor como parte do pedido. Os ataques refletidos são entregues às vítimas por meio de outra rota, como em uma mensagem de email ou em algum outro site. Quando um usuário é enganado a clicar em um link malicioso, enviando uma forma especialmente criada, ou mesmo apenas navegando para um site malicioso, o O código injetado viaja para o site vulnerável, que reflete o ataque de volta ao navegador do usuário.

Exemplo da vulnerabilidade Reflected XSS:

Na imagem a seguir o método Lambda incorpora dados não confiáveis na saída gerada com JSON no trecho em vermelho. Esses dados não confiáveis são incorporados à saída sem a devida higienização ou codificação, permitindo que um invasor injete código mal-intencionado na página da Web gerada.

O invasor seria capaz de alterar a página da Web retornada simplesmente fornecendo dados modificados no nome de entrada do usuário que é lido pelo método Lambda no trecho em amarelo. Essa entrada então flui através do código diretamente para a página da Web de saída, sem higienização. Isso pode habilitar um ataque XSS (Reflected Cross Site Scripting).

```
app.get('/vulnerabilidades/novafiltragem/:nome', (request, response) => {  
  let nome = request.params.nome;  
  
  // Construa a consulta SQL com segurança para evitar SQL injection  
  const query = `SELECT * FROM tbl_vulnerabilidades WHERE nome LIKE '%${nome}%'`;   
  
  db.query(query, (err, result) => {  
    if (err) {  
      console.error('Erro ao tentar retornar os dados:', err);  
      return response.status(500).json({ success: false, message: 'Erro na busca pelos dados na tabela de vulnerabilidades' });  
    }  
  
    if (result.length > 0) {  
      console.log('Dados de tabela de vulnerabilidades retornados!');  
      return response.json({ success: true, userInfo: result });  
    }  
  
  });  
  response.json(query)  
});
```

Como surge DOM XSS?

XSS baseado em DOM (ou como é chamado em alguns textos, "type-0 XSS") é um ataque XSS em que a carga útil do ataque é executada como resultado da modificação do "ambiente" DOM no navegador da vítima usado pelo script do lado do cliente original, para que o código do lado do cliente funciona de forma "inesperada". Ou seja, a própria página (o HTTP resposta que é) não muda, mas o código do lado do cliente contido em A página é executada de forma diferente devido às modificações maliciosas que ocorreram no ambiente DOM.

Exemplo da vulnerabilidade DOM XSS:

Na imagem a seguir o método Lambda incorpora dados não confiáveis na saída gerada com innerHTML, no trecho em vermelho. Esses dados não confiáveis são incorporados à saída sem a devida higienização ou codificação, permitindo que um invasor injete código mal-intencionado na página da Web gerada.

```
function buscarVulnerabilidades(nome) {
  // Verifique se o nome não está vazio
  if (!nome.trim()) {
    console.log('O campo de busca está vazio.');
```

```
    return;
  }

  fetch(`http://localhost:3000/vulnerabilidades?nome=${nome}`)
    .then(res => res.json())
    .then((json) => {
      console.log(json);

      const cards = document.getElementById('cards');
      cards.innerHTML = "";

      if (json.success) {
        json.userInfo.forEach((item) => {
          const card = document.createElement("div");
          card.classList.add("card");

          const cardContent = `
            <div href="http://localhost:3000/vulnerabilidades/${item.id}">
              <span class="item-name">${item.nome}</span>
              <p>${item.descricao}</p>
              <p class="item-referencia"> Referência:
              <a href="${item.link}" target="_blank" class="item-link"> ${item.link}</a>
            </div>
          `;

          cardContent.innerHTML = cardContent;
          card.innerHTML = cardContent;

          cards.appendChild(card);
        });
      } else {
        console.log('Dados da tabela de vulnerabilidades não encontrados');

        const mensagem = document.createElement("p");
        mensagem.innerText = "Nenhum dado encontrado na tabela de vulnerabilidades.";
        cards.appendChild(mensagem);

        cards.innerHTML = `${input.value}`;
      }
    })
    .catch(error => console.error('Erro na requisição:', error));
}
```

Recomendações gerais:

Codifique totalmente todos os dados dinâmicos, independentemente da origem, antes de incorporá-los na saída.

Por exemplo:

- Codificação HTML para conteúdo HTML
- Codificação de atributo HTML para saída de dados para valores de atributo
- Codificação JavaScript para JavaScript gerado pelo servidor
- É recomendável usar a funcionalidade de codificação fornecida pela plataforma ou bibliotecas de segurança conhecidas para a saída da codificação.
- Implemente uma Política de Segurança de Conteúdo (CSP) com listas brancas explícitas apenas para os recursos do aplicativo.

Como uma camada extra de proteção, valide todos os dados não confiáveis, independentemente da origem (observe que isso não é um substituto para a codificação), a validação deve ser baseada em uma lista branca: aceite apenas dados que se ajustam a uma estrutura especificada, em vez de rejeitar padrões incorretos.

Confira:

- Tipo de dado
- Tamanho
- Gama
- Formato
- Valores esperados
- No cabeçalho de resposta HTTP 'Content-Type', defina explicitamente a codificação de caracteres (charset) para a página inteira.
- Defina o sinalizador 'HTTPOnly' no cookie de sessão para "Defesa em profundidade", para evitar que explorações XSS bem-sucedidas roubem o cookie.

Links para um estudo mais aprofundado das vulnerabilidades XSS (Cross-Site Scripting):

<https://owasp.org/www-community/attacks/xss/>

https://owasp.org/www-community/attacks/DOM_Based_XSS

Vulnerabilidades SQL Injection (Injeção de SQL)-->

Visão geral:

Um ataque de [injeção de SQL](#) consiste em inserção ou "injeção" de uma consulta SQL através dos dados de entrada do cliente para o aplicação. Uma exploração de injeção de SQL bem-sucedida pode ler dados confidenciais do banco de dados, modificar dados do banco de dados (Inserir/Atualizar/Excluir), executar operações de administração no banco de dados (como desligar o SGBD), recuperar o conteúdo de um determinado arquivo presente no sistema de arquivos DBMS e Em alguns casos, emita comandos para o sistema operacional. Injeção de SQL são um tipo de ataque de injeção, no qual comandos SQL são injetados na entrada do plano de dados para afetar a execução de comandos SQL predefinidos.

Como surge SQL Injection?

O aplicativo armazena e gerencia dados em um banco de dados, enviando uma consulta SQL textual ao mecanismo de banco de dados para processamento. O aplicativo cria a consulta por concatenação de cadeia de caracteres simples, incorporando dados não confiáveis. No

entanto, não há separação entre dados e código; Além disso, os dados incorporados não são verificados quanto à validade do tipo de dados nem posteriormente higienizados. Assim, os dados não confiáveis podem conter comandos SQL ou modificar a consulta pretendida. O banco de dados interpretaria a consulta e os comandos alterados como se tivessem se originado do aplicativo e os executaria de acordo.

Exemplo da vulnerabilidade SQL Injection:

Na imagem a seguir o método Lambda do aplicativo executa uma consulta SQL com consulta no trecho em vermelho. O aplicativo constrói essa consulta SQL incorporando uma cadeia de caracteres não confiável na consulta sem a limpeza adequada. A cadeia de caracteres concatenada é enviada ao banco de dados, onde é analisada e executada de acordo. Um invasor seria capaz de injetar sintaxe e dados arbitrários na consulta SQL, criando uma carga maliciosa e fornecendo-a por meio do login de entrada; essa entrada é então lida pelo método Lambda no trecho em Amarelo.

Essa entrada então flui através do código, para uma consulta e para o servidor de banco de dados – sem higienização. Isso pode habilitar um ataque de injeção SQL.

```
app.get('/login/:login/:senha', (req, res) => {  
  const login = req.params.login;  
  const senha = req.params.senha;  
  
  const query = `SELECT nome, id FROM tbl_user WHERE login = '${login}' AND senha = '${senha}'`;  
  
  armazenarLog(query);  
  
  db.query(query, (err, result) => {  
    if (result && result.length > 0) {  
      const userID = result[0].id;  
      const sessionID = generateSessionID(userID);  
  
      console.log("UserId: " + userID)  
      console.log("SessionID: " + sessionID)  
      res.json({ success: true, sessionID, userInfo: { nome: login } });  
    } else {  
      res.json({ success: false, message: "Login failed" });  
    }  
  });  
});
```

Como surge Second Order SQL Injection?

O Ataque de Injeção SQL de Segunda Ordem insere código de linguagem SQL em solicitações de dados, fazendo com que o servidor de banco de dados de back-end do aplicativo entregue dados secretos ou execute material de programação mal-intencionado no banco de dados, potencialmente levando à penetração total do host. A entrada de injeção fornecida pelo usuário mal-intencionado será salva em uma injeção SQL de segunda ordem no banco de dados. Em seguida, ele é utilizado (sem limpeza suficiente) em uma nova consulta SQL quando um usuário visita outro recurso do mesmo aplicativo.

Exemplo da vulnerabilidade Second Order SQL Injection:

Na imagem a seguir o método Lambda do aplicativo executa uma consulta SQL com consulta, no trecho em vermelho. O aplicativo constrói essa consulta SQL incorporando uma cadeia de caracteres não confiável na consulta sem a limpeza adequada. A cadeia de caracteres concatenada é enviada ao banco de dados, onde é analisada e executada de acordo. O invasor pode gravar dados arbitrários no banco de dados, que são recuperados pelo aplicativo com resultado no método Lambda no trecho em vermelho.

Esses dados, em seguida, fluem através do código, até que sejam usados diretamente na consulta SQL sem limpeza e, em seguida, enviados ao servidor de banco de dados. Isso pode habilitar um ataque de injeção de SQL de segunda ordem.

```
app.get('/vulnerabilidades/novafiltragem/:nome', (request, response) => {  
  let nome = request.params.nome;  
  
  const query = `SELECT * FROM tbl_vulnerabilidades WHERE nome LIKE '%${nome}%'`;   
  
  db.query(query, (err, result) => {  
    if (err) {  
      console.error('Erro ao tentar retornar os dados:', err);  
      return response.status(500).json({ success: false, message: 'Erro na busca pelos dados na tabela de vulnerabilidades' });  
    }  
  
    if (result.length > 0) {  
      console.log('Dados de tabela de vulnerabilidades retornados!');  
      return response.json({ success: true, userInfo: result });  
    }  
  });  
  
  response.json(query)  
});
```

Recomendações gerais:

Como evitá-lo?

Validar todos os dados não confiáveis, independentemente da origem, A validação deve ser baseada em uma lista branca: aceite apenas dados que se ajustem a uma estrutura especificada, em vez de rejeitar padrões incorretos.

Em particular, verifique:

- Tipo de dado
- Tamanho
- Gama
- Formato
- Valores esperados
- Restrinja o acesso a objetos e funcionalidades do banco de dados, de acordo com o Princípio do Privilégio Mínimo.
- Não use cadeias de caracteres concatenadas dinamicamente para construir consultas SQL.
- Prefira usar procedimentos armazenados de banco de dados para todo o acesso a dados, em vez de consultas dinâmicas ad-hoc.
- Em vez de concatenação de cadeia de caracteres não segura, use componentes de banco de dados seguros, como consultas parametrizadas e associações de objetos (por exemplo, comandos e parâmetros).
- Como alternativa, uma solução ainda melhor é usar uma biblioteca ORM, a fim de predefinir e encapsular os comandos permitidos habilitados para o aplicativo, em vez de acessar dinamicamente o banco de dados diretamente. Dessa forma, o plano de código e o plano de dados devem ser isolados um do outro.

Links para um estudo mais aprofundado das vulnerabilidades SQL Injection (Injeção SQL):

https://owasp.org/www-community/attacks/SQL_Injection

<https://offensive360.com/second-order-sql-injection-attack/>

Vulnerabilidade Command Injection (Injeção de comando) -->

Visão geral:

A injeção de comando é um ataque em que o objetivo é a execução de comandos arbitrários no sistema operacional host por meio de uma aplicação vulnerável. Ataques de injeção de comando são possíveis quando um aplicativo passa dados fornecidos pelo usuário inseguros (formulários, cookies, cabeçalhos HTTP etc.) para um shell do sistema. Neste ataque, o

sistema operacional fornecido pelo invasor. Os comandos geralmente são executados com os privilégios dos vulneráveis aplicação. Os ataques de injeção de comando são possíveis em grande parte devido a validação de entrada insuficiente.

Como surge Command Injection?

O aplicativo executa um comando no nível do sistema operacional para concluir sua tarefa, em vez de por meio do código do aplicativo. O comando inclui dados não confiáveis, que podem ser controláveis por um invasor. Essa cadeia de caracteres não confiável pode conter comandos mal-intencionados no nível do sistema criados por um invasor, que podem ser executados como se o invasor estivesse executando comandos diretamente no servidor de aplicativos. Nesse caso, o aplicativo recebe dados da entrada do usuário e os passa como uma cadeia de caracteres para o sistema operacional. Esses dados não validados são então executados pelo sistema operacional como um comando do sistema, sendo executados com os mesmos privilégios de sistema que o aplicativo.

Exemplo da vulnerabilidade Command Injection:

Na imagem a seguir o método Lambda do aplicativo chamada um comando do sistema operacional (shell) com comando no trecho em vermelho, usando uma cadeia de caracteres não confiável com o comando a ser executado.

Isso pode permitir que um invasor injete um comando arbitrário e habilite um ataque de injeção de comando. O invasor pode ser capaz de injetar o comando executado por meio da entrada do usuário, filename, que é recuperado pelo aplicativo no método Lambda, no trecho em amarelo.

```
app.get('/executeCat/:filename', (req, res) => {
  const filename = req.params.filename;
  const command = `/usr/bin/cat ${filename}`;

  exec(command, (error, stdout, stderr) => {
    if (error) {
      console.error(`Erro ao executar o comando: ${error.message}`);
      return res.status(500).json({ success: false, message: 'Erro ao executar o comando' });
    }

    if (stderr) {
      console.error(`Erro no STDERR: ${stderr}`);
      return res.status(500).json({ success: false, message: 'Erro no STDERR do comando' });
    }

    console.log(`Comando executado com sucesso: ${stdout}`);
    res.json({ success: true, output: stdout });
  });
});
```

Recomendações gerais:

Como evitá-lo?

Refatore o código para evitar qualquer execução direta de comando do shell, Em vez disso, use APIs fornecidas pela plataforma ou chamadas de biblioteca, Se for impossível remover a execução do comando, execute apenas comandos estáticos que não incluam dados dinâmicos controlados pelo usuário, Validar todas as entradas, independentemente da fonte, A validação deve ser baseada em uma lista branca: aceitar apenas dados que se ajustam a um formato especificado, em vez de rejeitar padrões ruins (lista negra), Os parâmetros devem ser limitados a um conjunto de caracteres permitidos e a entrada não validada deve ser descartada, Além dos caracteres, verifique se: Tipo de dado Tamanho Gama Formato Valores esperados Para minimizar os danos como medida de defesa em profundidade, configure o aplicativo para ser executado usando uma conta de usuário restrita que não tenha privilégios desnecessários do sistema operacional, Se possível, isole todos os comandos do sistema operacional para usar uma conta de usuário dedicada separada que tenha privilégios mínimos apenas para os comandos e arquivos específicos usados pelo aplicativo, de acordo com o Princípio do Privilégio Mínimo.

Links para um estudo mais aprofundado da vulnerabilidade Command Injection (Injeção de comando):

https://owasp.org/www-community/attacks/Command_Injection

Vulnerabilidade Insecure Storage Of Sensitive Data (Armazenamento inseguro de dados confidenciais) -->

Visão geral:

O armazenamento inseguro de dados em um aplicativo móvel pode atrair vários agentes de ameaças que visam explorar as vulnerabilidades e obter acesso não autorizado a informações confidenciais. Esses agentes de ameaças incluem adversários qualificados que visam aplicativos móveis para extrair dados valiosos, insiders mal-intencionados dentro da organização ou da equipe de desenvolvimento de aplicativos que usam indevidamente seus privilégios, atores patrocinados pelo Estado que realizam espionagem cibernética, cibercriminosos que buscam ganhos financeiros por meio de roubo ou resgate de dados, script kiddies que utilizam ferramentas pré-construídas para ataques simples, corretores de dados que procuram explorar o armazenamento inseguro para vender informações pessoais, concorrentes e espiões industriais com o objetivo de obter uma vantagem competitiva, e ativistas ou hacktivistas com motivos ideológicos.

Como surge Insecure Storage Of Sensitive Data?

O aplicativo armazena informações confidenciais e pessoais, como PII, em sessões de usuário ou no banco de dados do aplicativo. Esses dados são armazenados em texto sem formatação, sem criptografia, permitindo que qualquer pessoa com acesso de leitura ao servidor roube esses segredos.

Exemplo da vulnerabilidade Insecure Storage Of Sensitive Data:

Na imagem a seguir observamos dados pessoais como senha, encontrados no trecho em amarelo, no qual são armazenados de forma desprotegida, sem criptografia, sendo usados para uma consulta no trecho em vermelho.

```
app.post('/register', (req, res) => {  
  const { login, nome, senha } = req.body;  
  const query = `INSERT INTO tbl_user (login, nome, senha) VALUES ('${login}', '${nome}', '${senha}')`;  
  
  db.query(query, (err, result) => {  
    if (err) {  
      console.error('Erro no registro:', err);  
      res.status(500).json({ success: false, message: 'Erro no registro' });  
    } else {  
      console.log('Usuário registrado com sucesso');  
      res.json({ success: true, message: 'Usuário registrado com sucesso' });  
    }  
  });  
});
```

Recomendações gerais:

Como evitá-lo?

- Não armazene dados pessoais ou outros segredos em texto simples, sem criptografia. Isso se aplica tanto ao armazenamento de longo prazo, como um banco de dados, quanto à memória de médio prazo, como sessões do lado do servidor.
- Os dados sensíveis devem ser sempre armazenados de forma criptografada, usando algoritmos de criptografia modernos (por exemplo, AES) e uma chave de criptografia suficientemente longa (por exemplo, 256 bits). As chaves de criptografia também devem ser protegidas.
- Como alternativa, alguns tipos de dados não devem ser armazenados em um formato reversível e podem ser hashed usando um algoritmo de hash criptograficamente forte, como SHA-256.

Link para um estudo mais aprofundado da vulnerabilidade Insecure Storage Of Sensitive Data (Armazenamento inseguro de dados confidenciais):

<https://owasp.org/www-project-mobile-top-10/2023-risks/m9-insecure-data-storage>

Vulnerabilidade XML External Entity (XXE) -->

Visão geral:

Um ataque de entidade externa XML é um tipo de ataque contra um aplicativo que analisa a entrada XML. Esse ataque ocorre quando **a entrada XML contendo uma referência a uma entidade externa é processado por um fraco analisador XML configurado**. Este ataque pode levar à divulgação de dados confidenciais, negação de serviço, falsificação de solicitação do lado do servidor, porta digitalização da perspectiva da máquina onde o analisador está e outros impactos no sistema.

Como surge XML External Entity (XXE)?

Um invasor pode carregar um documento XML que contenha uma declaração DTD, em particular uma definição de entidade que se refira a um arquivo local no disco do servidor, por exemplo, '`<! ENTIDADE xxe SISTEMA "file:///c:/boot.ini">`'. O invasor incluiria uma referência de entidade XML que se refere a essa definição de entidade, por exemplo, '`<div>&xxe; </div>`'. Se o documento XML analisado for retornado ao usuário, o resultado incluirá o conteúdo do arquivo de sistema confidencial. Isso é causado pelo analisador XML, que é configurado para analisar automaticamente declarações DTD e resolver referências de entidade, em vez de desabilitar DTD e referências externas completamente.

Exemplo da vulnerabilidade XML External Entity (XXE):

Na imagem a seguir o Lambda carrega e analisa XML usando `parseXml`, no trecho em vermelho. Esse XML foi recebido anteriormente da entrada do usuário, `xmlObj`, no trecho em amarelo. Observe que `parseXml` é definido para carregar e substituir automaticamente quaisquer referências de entidade DTD no XML, incluindo referências a arquivos externos.

```

app.post('/xml-data', (req, res) => {
  try {
    const xmlObj = req.body.xml;
    if (!xmlObj || typeof xmlObj !== 'string') {
      return res.status(400).json({ success: false, message: 'O parâmetro "xml" é obrigatório e deve ser uma string para a inserção.' });
    }

    try {
      const xmlDoc = libxmljs.parseXml(xmlObj.toString(), {
        noent: true,
      });

      console.log(xmlDoc.toString());
      const timestamp = new Date().getTime();
      const randomPart = Math.floor(Math.random() * 1000);
      const fileName = `arquivo_${timestamp}_${randomPart}.xml`;

      const xmlString = xmlDoc.toString();

      fs.writeFile(fileName, xmlString, (err) => {
        if (err) {
          console.error('Erro ao escrever o arquivo:', err);
          return res.status(500).json({ success: false, message: 'Erro ao escrever o arquivo' });
        }

        res.status(200).json({ success: true, fileName });
      });
    } catch (error) {
      console.error('Erro ao analisar XML:', error);
      res.status(500).json({ success: false, message: 'Erro ao processar XML' });
    }
  } catch (error) {
    console.error('Erro no servidor:', error);
    res.status(500).json({ success: false, message: 'Erro interno no servidor', error: error.message });
  }
});

```

Recomendações gerais:

Como evitá-lo?

Orientações genéricas:

- Evite processar a entrada do usuário diretamente, sempre que possível.
- Se necessário para receber XML do usuário, verifique se o analisador XML está restrito e restrito.
- Em particular, desabilite a análise e a resolução de DTD de entidades. Aplique um esquema XML estrito no servidor e valide o XML de entrada de acordo.

Recomendações específicas:

- Use analisadores XML seguros e desabilite a análise DTD e a resolução de entidades.
- Não habilite a análise de DTD ou a resolução de entidades.
-

Link para um estudo mais aprofundado da vulnerabilidade XML External Entity (XXE):

[https://owasp.org/www-community/vulnerabilities/XML_External_Entity_\(XXE\)_Processing](https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing)

Vulnerabilidade Unchecked Input For Loop Condition (Entrada não verificada para condição de loop) -->

Visão geral:

O produto não verifica corretamente as entradas que são usadas para condições de loop, potencialmente levando a uma negação de serviço ou outras consequências devido ao looping excessivo.

Como surge Unchecked Input For Loop Condition (Entrada não verificada para condição de loop)?

O aplicativo executa alguma tarefa repetitiva em um loop e define o número de vezes para executar o loop de acordo com a entrada do usuário. Um valor muito alto pode fazer com que o aplicativo fique preso no loop e não possa continuar com outras operações.

Exemplo da vulnerabilidade Unchecked Input For Loop Condition (Entrada não verificada para condição de loop):

Na imagem o método no trecho em vermelho, obtém entrada do usuário do elemento headBuf, o valor deste elemento flui pelo código sem ser validado e é eventualmente usado em uma condição em loop no trecho em amarelo, isso constitui uma entrada não verificada para Loop.

```
POSITION: for (position = 0; position < st.size; position += 512) {
  for (let bufPos = 0, bytes = 0; bufPos < 512; bufPos += bytes) {
    bytes = fs.readSync(
      fd, headBuf, bufPos, headBuf.length - bufPos, position + bufPos
    )

    if (position === 0 && headBuf[0] === 0x1f && headBuf[1] === 0x8b) {
      throw new Error('cannot append to compressed archives')
    }

    if (!bytes) {
      break POSITION
    }
  }

  const h = new Header(headBuf)
  if (!h.cksumValid) {
    break
  }
  const entryBlockSize = 512 * Math.ceil(h.size / 512)
  if (position + entryBlockSize + 512 > st.size) {
    break
  }
  // the 512 for the header we just parsed will be added as well
  // also jump ahead all the blocks for the body
  position += entryBlockSize
  if (opt.mtimeCache) {
    opt.mtimeCache.set(h.path, h.mtime)
  }
}
```


Recomendações gerais:

Como evitá-lo?

- Idealmente, não baseie um loop em dados fornecidos pelo usuário. Se for necessário fazê-lo, a entrada do usuário deve ser validada primeiro e seu alcance deve ser limitado.

Link para um estudo mais aprofundado da vulnerabilidade Unchecked Input For Loop Condition (Entrada não verificada para condição de loop):

<https://cwe.mitre.org/data/definitions/606.html>

Vulnerabilidade Stored Code Injection (injeção de código armazenado) -->

Visão geral:

Code Injection é o termo geral para tipos de ataque que consistem em injetando código que é então interpretado/executado pelo aplicativo. Esse tipo de ataque explora o tratamento inadequado de dados não confiáveis. Estes tipos de ataques geralmente são possíveis devido à falta de validação de dados de entrada/saída, por exemplo:

- caracteres permitidos (expressões regulares padrão, classes ou personalizadas)
- formato de dados
- quantidade de dados esperados.

Como surge Stored Code Injection (injeção de código armazenado)?

O aplicativo executa alguma ação criando e executando código que inclui dados não confiáveis, que podem estar sob controle de um usuário mal-intencionado. Se os dados contiverem código mal-intencionado, o código executado poderá conter atividades no nível do sistema projetadas por um invasor, como se o invasor estivesse executando código diretamente no servidor de aplicativos.

Exemplo da vulnerabilidade Stored Code Injection (injeção de código armazenado):

Na imagem a seguir o método Lambda da aplicação recebe e executa dinamicamente o código controlado pelo usuário usando fileContent, no trecho em vermelho. Isso pode

permitir que um invasor injete e execute código arbitrário, o atacante pode injetar o código executado inserindo a carga útil em campos de texto padrão no banco de dados ou em um arquivo local, este texto é recuperado pelo método Lambda, usando `fileContent` no trecho em amarelo, e em seguida, enviado para o método de execução.

```
app.post('/armazenamento/arquivo', (req, res) => {
  const { nome_arquivo, conteudo_arquivo } = req.body;

  if (!nome_arquivo || !conteudo_arquivo) {
    return res.status(400).json({ success: false, message: 'Os parâmetros "nome_arquivo" e "conteudo_arquivo" são obrigatórios para a inserção.' });
  }

  const timestamp = new Date().getTime();
  const randomPart = Math.floor(Math.random() * 1000);
  const fileName = `arquivo_${timestamp}_${randomPart}.xml`;

  const query = 'INSERT INTO tbl_armazenamento_arquivos (nome_arquivo, conteudo_arquivo) VALUES (?, ?)';

  fs.appendFile(fileName, conteudo_arquivo, (err) => {
    if (err) {
      console.error('Erro ao escrever o arquivo:', err);
      return res.status(500).send('Erro ao escrever o arquivo');
    }
  });

  fs.readFile(`../gustavo_projeto_webgoat_back_end/${fileName}`, (err, fileContent) => {
    if (err) {
      console.error('Erro ao ler o arquivo:', err);
      return res.status(500).send('Erro ao ler o arquivo');
    }

    try {
      eval(fileContent);

      res.writeHead(200, {'Content-Type': 'text/xml'});
      res.write(fileContent);

      console.log('O que está dentro do conteúdo: ' + fileContent);

      db.query(query, [nome_arquivo, fileContent], (err, result) => {
        if (err) {
          console.error('Erro ao tentar inserir dados:', err);
        } else {
          console.log('Dados inseridos com sucesso!');
        }
      });
    } catch (error) {
      console.error('Erro ao executar o script:', error);
      return res.status(500).send('Erro ao executar o script');
    }
  });
});
});
```

Recomendações gerais:

Como evitá-lo?

- O aplicativo não deve compilar, executar ou avaliar qualquer código não confiável de qualquer fonte externa, incluindo entrada do usuário, arquivos carregados ou um banco de dados.
- Se for absolutamente necessário incluir dados externos na execução dinâmica, é permitido passar os dados como parâmetros para o código, mas não executar dados do usuário diretamente.
- Se for necessário passar dados não confiáveis para a execução dinâmica, imponha uma validação de dados muito rigorosa. Por exemplo, aceite apenas inteiros entre determinados valores.

Validar todas as entradas, independentemente da fonte, A validação deve ser baseada em uma lista branca: aceite apenas dados que ajustem uma estrutura especificada, em vez de rejeitar padrões incorretos, os parâmetros devem ser limitados a um conjunto de caracteres permitidos e a entrada não validada deve ser descartada.

Além dos caracteres, verifique:

- Tipo de dado
- Tamanho
- Gama
- Formato
- Valores esperados
- Se possível, prefira sempre colocar na lista de permissões entradas conhecidas e confiáveis em vez de comparar com uma lista negra.
- Configure o aplicativo para ser executado usando uma conta de usuário restrita que não tenha privilégios desnecessários.
- Se possível, isole toda a execução dinâmica para usar uma conta de usuário separada e dedicada que tenha privilégios apenas para as operações e arquivos específicos usados pela execução dinâmica, de acordo com o Princípio do Privilégio Mínimo.
- Prefira passar dados do usuário para um script pré-implementado, por exemplo, em outro aplicativo isolado, em vez de executar dinamicamente o código controlado pelo usuário.

Link para um estudo mais aprofundado da vulnerabilidade Stored Code Injection (injeção de código armazenado):

https://owasp.org/www-community/attacks/Code_Injection

Vulnerabilidade Server DoS by sleep (Servidor Negação de serviço por suspensão) -->

Visão geral:

O ataque de negação de serviço (DoS) é focado em criar um recurso (site, aplicativo, servidor) indisponível para a finalidade que foi projetada. Há muitas maneiras de tornar um serviço indisponível para usuários legítimos manipulando pacotes de rede, programação, lógica ou recursos lidar com vulnerabilidades, entre outras. Se um serviço recebe um grande número de pedidos, pode deixar de estar disponível para legítimos Usuários. Da mesma forma, um serviço pode parar se uma programação vulnerabilidade é explorada, ou a maneira como o serviço lida com recursos Usa.

Como surge Server DoS by sleep (Servidor Negação de serviço por suspensão)?

O aplicativo usa um valor fornecido pelo usuário para definir seu período de suspensão, sem impor um intervalo limitado para esse valor.

Exemplo da vulnerabilidade Server DoS by sleep (Servidor Negação de serviço por suspensão):

Na imagem a seguir o método Lambda no trecho em vermelho, obtém a entrada do usuário para o elemento url, o valor deste elemento é eventualmente usado para definir o período de “sleep” da aplicação, em Lambda no trecho em amarelo. Isso pode permitir um ataque DoS por Sleep.

```
if (request.timeout) {
  req.once('socket', function (socket) {
    reqTimeout = setTimeout(function () {
      reject(new FetchError('network timeout at: ${request.url}', 'request-timeout'));
      finalize();
    }, request.timeout);
  });
}

req.on('error', function (err) {
  reject(new FetchError('request to ${request.url} failed, reason: ${err.message}', 'system', err));

  if (response && response.body) {
    destroyStream(response.body, err);
  }

  finalize();
});

fixResponseChunkedTransferBadEnding(req, function (err) {
  if (signal && signal.aborted) {
    return;
  }

  if (response && response.body) {
    destroyStream(response.body, err);
  }
});

if (parseInt(process.version.substring(1)) < 14) {
  req.on('socket', function (s) {
    s.addListener('close', function (hadError) {
      const hasDataListener = s.listenerCount('data') > 0;

      if (response && hasDataListener && !hadError && !(signal && signal.aborted)) {
        const err = new Error('Premature close');
        err.code = 'ERR_STREAM_PREMATURE_CLOSE';
        response.body.emit('error', err);
      }
    });
  });
}

req.on('response', function (res) {
  clearTimeout(reqTimeout);

  const headers = createHeadersLenient(res.headers);

  if (fetch.isRedirect(res.statusCode)) {
    const location = headers.get('Location');

    let locationURL = null;
    try {
      locationURL = location === null ? null : new URL(location, request.url).toString();
    } catch (err) {
      if (request.redirect !== 'manual') {
        reject(new FetchError(`uri requested responds with an invalid redirect URL: ${location}`, 'invalid-redirect'));
        finalize();
        return;
      }
    }
  }
}
```

Recomendações gerais:

Como evitá-lo?

- Idealmente, a duração do comando sleep não deve ser de acordo com a entrada do usuário. Ele deve ser codificado, definido em um arquivo de configuração ou calculado dinamicamente em tempo de execução.
- Se for necessário permitir que o usuário defina a duração do sono, este valor DEVE ser verificado e imposto para estar dentro de um intervalo predefinido de valores válidos.

Como surge Server DoS by loop(Servidor Negação de serviço por loop)?

O aplicativo dependia de um valor fornecido pelo usuário para determinar o número de iterações executadas por um loop, sem impor um intervalo limitado para esse valor.

Exemplo da vulnerabilidade Server DoS by loop (Servidor Negação de serviço por loop):

A aplicação entra em um loop com n no trecho em vermelho. No entanto, para determinar a quantidade de iterações que este loop realiza, a aplicação depende dos cabeçalhos de entrada do usuário no trecho em amarelo.

```
function configureAllowedHeaders(options, req) {
  var allowedHeaders = options.allowedHeaders || options.headers;
  var headers = [];

  if (!allowedHeaders) {
    allowedHeaders = req.headers['access-control-request-headers']; // .headers wasn't specified, so reflect the request headers
    headers.push([
      key: 'Vary',
      value: 'Access-Control-Request-Headers'
    ]);
  } else if (allowedHeaders.join) {
    allowedHeaders = allowedHeaders.join(','); // .headers is an array, so turn it into a string
  }
  if (allowedHeaders && allowedHeaders.length) {
    headers.push([
      key: 'Access-Control-Allow-Headers',
      value: allowedHeaders
    ]);
  }
  return headers;
}

function configureExposedHeaders(options) {
  var headers = options.exposedHeaders;
  if (!headers) {
    return null;
  } else if (headers.join) {
    headers = headers.join(','); // .headers is an array, so turn it into a string
  }
  if (headers && headers.length) {
    return {
      key: 'Access-Control-Expose-Headers',
      value: headers
    };
  }
  return null;
}

function configureMaxAge(options) {
  var maxAge = (typeof options.maxAge === 'number' || options.maxAge) && options.maxAge.toString()
  if (maxAge && maxAge.length) {
    return {
      key: 'Access-Control-Max-Age',
      value: maxAge
    };
  }
  return null;
}

function applyHeaders(headers, res) {
  for (var i = 0, n = headers.length; i < n; i++) {
    var header = headers[i];
    if (header) {
      if (Array.isArray(header)) {
        applyHeaders(header, res);
      } else if (header.key === 'Vary' && header.value) {
        vary(res, header.value);
      } else if (header.value) {
        res.setHeader(header.key, header.value);
      }
    }
  }
}
```

Recomendações gerais:

Como evitá-lo?

- Considere evitar a dependência das entradas do usuário para determinar o número de iterações em um loop
- Quando uma conta de iteração dinâmica baseada na entrada do usuário for necessária, sempre restrinja a quantidade de iterações e tolere ou falhe graciosamente se essa contagem de iterações for excedida

Link para um estudo mais aprofundado da vulnerabilidade Server DoS by sleep (Servidor Negação de serviço por suspensão):

https://owasp.org/www-community/attacks/Denial_of_Service

Vulnerabilidade Sensitive information Over HTTP (Informações confidenciais por HTTP) -->

Visão geral:

A exposição de informações por meio de cadeias de caracteres de consulta na URL ocorre quando dados confidenciais são passados para parâmetros na URL. Isso permite que os invasores obtenham dados confidenciais, como nomes de usuário, senhas, tokens (authX), detalhes do banco de dados e quaisquer outros dados potencialmente confidenciais. O simples uso de HTTPS não resolve essa vulnerabilidade.

Como surge Sensitive information Over HTTP (Informações confidenciais por HTTP)?

O aplicativo manipula várias formas de dados confidenciais e se comunica com o servidor de aplicativos remoto. No entanto, o aplicativo se conecta usando uma URL "http://", o que fará com que o canal subjacente use HTTP direto, sem protegê-lo com SSL/TLS.

Exemplo da vulnerabilidade Sensitive information Over HTTP (Informações confidenciais por HTTP):

Na imagem a seguir o método de registro do aplicativo no trecho em amarelo, envia uma solicitação HTTP para o servidor usando fetch, no entrando, essa solicitação é enviada por

HTTP não protegido, sem garantir o canal com HTTPS. Isso pode expor informações sensíveis como a senha do registro no trecho em vermelho.

```
function register() {  
  const login = document.getElementById('registerLogin').value;  
  const nome = document.getElementById('registerName').value;  
  const senha = document.getElementById('registerPassword').value;  
  
  fetch('http://localhost:3000/register', {  
    method: 'POST',  
    headers: {  
      'Content-Type': 'application/json',  
    },  
    body: JSON.stringify({ login, nome, senha }),  
  })  
  .then(response => response.json())  
  .then(data => {  
    if (data.success) {  
      console.log("User registered");  
    } else {  
      console.error("Registration failed:", data.message);  
    }  
  })  
  .catch(error => {  
    console.error("Error:", error);  
  
    const errorMessage = document.createElement("p");  
    errorMessage.innerHTML = `Error: ${error}`;  
    document.body.appendChild(errorMessage);  
  });  
}
```

Recomendações gerais:

Como evitá-lo?

- Sempre use protocolo seguro conectando-se a URLs "https://", Nunca envie dados confidenciais para uma URL "http://".

Link para um estudo mais aprofundado da vulnerabilidade Sensitive information Over HTTP (Informações confidenciais por HTTP):

[https://owasp.org/www-community/vulnerabilities/Information exposure through query strings in url#:~:text=Information%20exposure%20through%20query%20strings%20in%20URL%20is,Simply%20using%20HTTPS%20does%20not%20resolve%20this%20vulnerability.](https://owasp.org/www-community/vulnerabilities/Information_exposure_through_query_strings_in_url#:~:text=Information%20exposure%20through%20query%20strings%20in%20URL%20is,Simply%20using%20HTTPS%20does%20not%20resolve%20this%20vulnerability.)

Vulnerabilidade Privacy Violation (Violação de privacidade) -->

Visão geral:

Tratamento indevido de informações privadas, como senhas de clientes ou redes sociais números de segurança, pode comprometer a privacidade do usuário, e muitas vezes é ilegal.

Como surge Privacy Violation (Violação de privacidade)?

O aplicativo envia informações do usuário, como senhas, informações de conta ou números de cartão de crédito, para fora do aplicativo, como gravá-las em um arquivo de texto ou de log local ou enviá-las para um serviço Web externo.

Exemplo da vulnerabilidade Privacy Violation (Violação de privacidade):

Na imagem a seguir o método `_instructions` no trecho em vermelho, envia informações do usuário para fora do aplicativo. Isso pode constituir uma violação de privacidade.

```
function _instructions (result, dotenvKey) {
  // Parse DOTENV_KEY. Format is a URI
  let uri
  try {
    uri = new URL(dotenvKey)
  } catch (error) {
    if (error.code === 'ERR_INVALID_URL') {
      throw new Error('INVALID_DOTENV_KEY: Wrong format. Must be in valid uri format like dotenv:///key_1234@dotenv.org/vault/.env.vault?environment=development')
    }
    throw error
  }

  // Get decrypt key
  const key = uri.password
  if (!key) {
    throw new Error('INVALID_DOTENV_KEY: Missing key part')
  }

  // Get environment
  const environment = uri.searchParams.get('environment')
  if (!environment) {
    throw new Error('INVALID_DOTENV_KEY: Missing environment part')
  }

  // Get ciphertext payload
  const environmentKey = `DOTENV_VAULT_${environment.toUpperCase()}`
  const ciphertext = result.parsed[environmentKey] // DOTENV_VAULT_PRODUCTION
  if (!ciphertext) {
    throw new Error('NOT_FOUND_DOTENV_ENVIRONMENT: Cannot locate environment ${environmentKey} in your .env.vault file.')
  }

  return { ciphertext, key }
}
```

Como surge Client Privacy Violation (Violação de privacidade do cliente)?

O aplicativo envia informações do usuário, como senhas, informações de conta ou números de cartão de crédito, para fora do aplicativo, como gravá-las em um arquivo de texto ou de log local ou enviá-las para um serviço Web externo.

Exemplo da vulnerabilidade Client Privacy Violation (Violação de privacidade do cliente):

Na imagem a seguir o login do método no trecho em vermelho, envia informações do usuário para fora do aplicativo. Isso pode construir uma violação de privacidade.

```
function login() {
  const login = document.getElementById('loginLogin').value;
  const senha = document.getElementById('loginSenha').value;
  const url = `http://localhost:3000/login/${login}/${senha}`;

  fetch(url)
    .then(response => response.json())
    .then(data => {
      if (data.success) {
        console.log("User logged in:", data.userInfo);

        document.cookie = `sessionID=${data.sessionID}; path=/`;
        console.log("Session ID:", data.sessionID);

        window.location.href = 'filtrar_vulnerabilidades.html';
      } else {
        console.error("Login failed:", data.message);
      }
    })
    .catch(error => console.error("Error:", error));
}
```

Recomendações gerais:

Como evitá-lo?

- Os dados pessoais devem ser removidos antes de serem gravados em logs ou outros arquivos.
- Analisar a necessidade e a justificativa do envio de dados pessoais para serviços web remotos.

Link para um estudo mais aprofundado da vulnerabilidade Privacy Violation (Violação de privacidade):

https://owasp.org/www-community/vulnerabilities/Privacy_Violation

Vulnerabilidade Missing HSTS Header (Cabeçalho HSTS ausente) -->

Visão geral:

HTTP Strict Transport Security. (também chamado **HSTS**) é um aprimoramento de segurança opcional que é especificado por um aplicativo Web por meio do uso de um cabeçalho de resposta especial. Assim que um navegador suportado receber esse cabeçalho, esse navegador impedirá que quaisquer comunicações sejam enviadas por HTTP para o domínio

especificado e, em vez disso, enviará todas as comunicações por HTTPS. Ele também impede que o HTTPS clique em prompts em navegadores.

Como surge Missing HSTS Header (Cabeçalho HSTS ausente)?

Muitos usuários navegam para sites simplesmente digitando o nome de domínio na barra de endereço, sem o prefixo do protocolo. O navegador assumirá automaticamente que o protocolo pretendido do usuário é HTTP, em vez do protocolo HTTPS criptografado. Quando essa solicitação inicial é feita, um invasor pode executar um ataque Man-in-the-Middle e manipulá-lo para redirecionar os usuários para um site mal-intencionado de sua escolha. Para proteger o usuário de tal ocorrência, o cabeçalho HTTP Strict Transport Security (HSTS) instrui o navegador do usuário a não permitir o uso de uma conexão HTTP não segura com o domínio associado ao cabeçalho HSTS. Depois que um navegador que suporta o recurso HSTS visitar um site e o cabeçalho tiver sido definido, ele não permitirá mais a comunicação com o domínio por meio de uma conexão HTTP.

Exemplo da vulnerabilidade Missing HSTS Header (Cabeçalho HSTS ausente)

Na imagem a seguir observe o trecho em vermelho. O aplicativo Web não define um cabeçalho HSTS, deixando-o vulnerável a ataques.

```
app.post('/processar-xml', (req, res) => {
  const xml = req.body;

  try {
    const xmlDoc = libxmljs.parseXmlString(xml, {
      noent: false, // Habilitar o processamento de entidades externas
      // Outras opções, se necessário
    });

    console.log(xmlDoc.toString());
    res.status(200).json({ success: true, message: 'XML processado com sucesso!' });
  } catch (error) {
    console.error('Erro ao analisar XML:', error);
    res.status(500).send('Erro ao processar XML');
  }
});
```

Recomendações gerais:

Como evitá-lo?

Antes de definir o cabeçalho HSTS - considere as implicações que ele pode ter: Forçar HTTPS impedirá qualquer uso futuro de HTTP, o que poderia dificultar alguns testes. Desabilitar o HSTS não é trivial, pois uma vez desativado no site, ele também deve ser desativado no navegador. Defina o cabeçalho HSTS explicitamente no código do aplicativo ou usando configurações de servidor Web. Certifique-se de que o valor de "idade máxima" para cabeçalhos HSTS esteja definido como 31536000 para garantir que o HSTS seja

rigorosamente aplicado por pelo menos um ano, Inclua o "includeSubDomains" para maximizar a cobertura do HSTS e garantir que o HSTS seja aplicado em todos os subdomínios sob o domínio atual Observe que isso pode impedir o acesso seguro do navegador a quaisquer subdomínios que utilizem HTTP; no entanto, o uso de HTTP é muito severo e altamente desencorajado, mesmo para sites que não contêm nenhuma informação sensível, pois seu conteúdo ainda pode ser adulterado por meio de ataques Man-in-the-Middle para usuários fanáticos sob o domínio HTTP.

Link para um estudo mais aprofundado da vulnerabilidade Missing HSTS Header (Cabeçalho HSTS ausente):

[https://cheatsheetseries.owasp.org/cheatsheets/HTTP Strict Transport Security Cheat Sheet.html#Introduction%C2%B6](https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html#Introduction%C2%B6)

Vulnerabilidade Client DOM Cookie Poisoning -->

Visão geral:

Em um ataque de *envenenamento* de cookies, o invasor manipula o conteúdo dos cookies HTTP antes que eles sejam entregues do navegador do usuário para um aplicativo da web. Os invasores podem preceder o envenenamento de cookies com o [sequestro de cookies](#), o que permite que eles obtenham acesso ao conteúdo do cookie antes de adulterá-lo, mas isso nem sempre é necessário.

Como surge Client DOM Cookie Poisoning?

O aplicativo não impede que entradas maliciosas sejam inseridas nos cookies do aplicativo. Um invasor pode influenciar o navegador da vítima com parâmetros de URL mal-intencionados, fazendo com que o script carregue esses valores mal-intencionados no cookie do usuário. Isso pode ser por meio de phishing, links armazenados, links externos e muito mais.

Exemplo da vulnerabilidade Missing HSTS Header (Cabeçalho HSTS ausente)

Na imagem a seguir a aplicação define um cookie, cookie, no método Lambda do arquivo no trecho em vermelho, o valor definido para este cookie é controlado pelo valor de entrada do usuário externo, no trecho em amarelo do arquivo. No método de login esta entrada pode ser controlada por um terceiro externo.

```
function login() {
  const login = document.getElementById('loginLogin').value;
  const senha = document.getElementById('loginSenha').value;
  const url = `http://localhost:3000/login/${login}/${senha}`;

  fetch(url)
    .then(response => response.json())
    .then(data => {
      if (data.success) {
        console.log("User logged in:", data.userInfo);

        document.cookie = `sessionID=${data.sessionID}; path=/`;
        console.log("Session ID:", data.sessionID);

        window.location.href = 'filtrar_vulnerabilidades.html';
      } else {
        console.error("Login failed:", data.message);
      }
    })
    .catch(error => console.error("Error:", error));
}
```

Recomendações gerais:

Como evitá-lo?

- Não defina cookies com base em entradas controláveis pelo usuário, como fragmentos de URL, parâmetros GET ou valores de campos de entrada.

Links para um estudo mais aprofundado da vulnerabilidade Client DOM

Cookie Poisoning:

<https://portswigger.net/web-security/dom-based/cookie-manipulation>

<https://www.invicti.com/learn/cookie-poisoning/>

Vulnerabilidade Absolute Path Traversal (Travessia Absoluta do Caminho) -->

Visão geral:

Um ataque de travessia de caminho (também conhecido como travessia de diretório) tem como objetivo Acessar arquivos e diretórios armazenados fora da raiz da Web pasta. Manipulando variáveis que fazem referência a arquivos com "ponto-ponto-barra (.. /)" e suas variações ou usando caminhos de arquivo absolutos, pode ser possível acessar arquivos arbitrários e diretórios armazenados no sistema de arquivos, incluindo o código-fonte do aplicativo ou configuração e arquivos críticos do sistema. Deve-se notar que o acesso aos arquivos é limitado pelo controle de acesso operacional do sistema (como no caso de arquivos bloqueados ou em uso no Microsoft Windows operacional sistema).

Este ataque também é conhecido como "dot-dot-slash", "directory traversal", "Directory climbing" e "backtracking".

Como surge Absolute Path Traversal (Travessia Absoluta do Caminho)?

O aplicativo usa a entrada do usuário no caminho do arquivo para acessar arquivos no disco local do servidor de aplicativos.

Exemplo da vulnerabilidade Absolute Path Traversal (Travessia Absoluta do Caminho):

Na imagem a seguir o método enviado no trecho em amarelo, obtém dados dinâmicos do elemento de intervalo, o valor deste elemento flui através do código e é eventualmente usado em um caminho de arquivo para acesso ao disco local em sendFile no trecho em vermelho. Isso pode causar uma vulnerabilidade de Traversal de Caminho.

```

SendStream.prototype.send = function send (path, stat) {
  var len = stat.size
  var options = this.options
  var opts = {}
  var res = this.res
  var req = this.req
  var ranges = req.headers.range
  var offset = options.start || 0

  if (headersSent(res)) {
    // impossible to send now
    this.headersAlreadySent()
    return
  }

  debug('pipe "%s"', path)
  this.setHeader(path, stat)
  this.type(path)

  if (this.isConditionalGET()) {
    if (this.isPreconditionFailure()) {
      this.error(412)
      return
    }

    if (this.isCachable() && this.isFresh()) {
      this.notModified()
      return
    }
  }

  len = Math.max(0, len - offset)
  if (options.end !== undefined) {
    var bytes = options.end - offset + 1
    if (len > bytes) len = bytes
  }

  if (this._acceptRanges && BYTES_RANGE_REGEXP.test(ranges)) {
    // parse
    ranges = parseRange(len, ranges, {
      combine: true
    })
  }

  if (!this.isRangeFresh()) {
    debug('range stale')
    ranges = -2
  }

  if (ranges === -1) {
    debug('range unsatisfiable')

    res.setHeader('Content-Range', contentRange('bytes', len))
    return this.error(416, {
      headers: { 'Content-Range': res.getHeader('Content-Range') }
    })
  }

  if (ranges !== -2 && ranges.length === 1) {
    debug('range %j', ranges)

    res.statusCode = 206
    res.setHeader('Content-Range', contentRange('bytes', len, ranges[0]))

    offset += ranges[0].start
    len = ranges[0].end - ranges[0].start + 1
  }

  for (var prop in options) {
    opts[prop] = options[prop]
  }

  opts.start = offset
  opts.end = Math.max(offset, offset + len - 1)

  res.setHeader('Content-Length', len)

  if (req.method === 'HEAD') {
    res.end()
    return
  }

  this.stream(path, opts)
}

SendStream.prototype.sendFile = function sendFile (path) {
  var i = 0
  var self = this

  debug('stat "%s"', path)
  fs.stat(path, function onstat (err, stat) {
    if (err && err.code === 'ENOENT' && !extname(path) && path[path.length - 1] !== sep) {
      // not found, check extensions
      return next(err)
    }
    if (err) return self.onStatError(err)
    if (stat.isDirectory()) return self.redirect(path)
    self.emit('file', path, stat)
    self.send(path, stat)
  })
}

```

Recomendações gerais:

Como evitá-lo?

- Idealmente, evitar depender de dados dinâmicos para a seleção de arquivos, 2, Validar todas as entradas, independentemente da fonte, A validação deve ser baseada em uma lista branca: aceite apenas dados que se ajustem a uma estrutura especificada, em vez de rejeitar padrões incorretos, Verificar para: Tipo de dado Tamanho Gama Formato Valores esperados 4, Aceitar dados dinâmicos apenas para o nome do arquivo, não para o caminho e pastas, 5, Certifique-se de que o caminho do arquivo seja totalmente canonicalizado, 6, Limite explicitamente o aplicativo para usar uma pasta designada que é separada da pasta binária de aplicativos, 7, Restrinja os privilégios do usuário do sistema operacional do aplicativo aos arquivos e pastas necessários, O aplicativo não deve ser capaz de gravar na pasta binária do aplicativo e não deve ler nada fora da pasta do aplicativo e da pasta de dados.

Link para um estudo mais aprofundado da vulnerabilidade Absolute Path Traversal (Travessia Absoluta do Caminho):

https://owasp.org/www-community/attacks/Path_Traversal

Vulnerabilidade Use Of Deprecated Or Obsolete Functions (Uso de funções depreciadas ou obsoletas) -->

Como surge Use Of Deprecated Or Obsolete Functions (Uso de funções depreciadas ou obsoletas)?

O aplicativo faz referência a elementos de código que foram declarados como preteridos. Isso pode incluir classes, funções, métodos, propriedades, módulos ou versões de biblioteca obsoletas que estão desatualizadas por versão ou foram totalmente preteridas.

Exemplo da vulnerabilidade Use Of Deprecated Or Obsolete Functions (Uso de funções depreciadas ou obsoletas):

Na imagem a seguir o método `fromArrayBuffer` no trecho em vermelho, chama uma API obsoleta, `Buffer`. Isso foi descontinuado e não deve ser usado em uma base de código moderna.

```
function fromArrayBuffer (obj, byteOffset, length) {  
  byteOffset >>= 0  
  
  var maxLength = obj.byteLength - byteOffset  
  
  if (maxLength < 0) {  
    throw new RangeError("'offset' is out of bounds")  
  }  
  
  if (length === undefined) {  
    length = maxLength  
  } else {  
    length >>= 0  
  
    if (length > maxLength) {  
      throw new RangeError("'length' is out of bounds")  
    }  
  }  
  
  return isModern  
    ? Buffer.from(obj.slice(byteOffset, byteOffset + length))  
    : new Buffer(new Uint8Array(obj.slice(byteOffset, byteOffset + length)))  
}
```

Recomendações gerais:

Como evitá-lo?

- Prefira sempre usar as versões mais atualizadas de bibliotecas, pacotes e outras dependências.
- Não use ou faça referência a qualquer classe, método, função, propriedade ou outro elemento que tenha sido declarado preterido.

Vulnerabilidade Use Of Broken Or Risky Cryptographic Algorithm (Uso de algoritmo criptográfico quebrado ou arriscado) -->

Como surge Use Of Broken Or Risky Cryptographic Algorithm (Uso de algoritmo criptográfico quebrado ou arriscado)?

O código do aplicativo especifica o nome do algoritmo criptográfico selecionado, por meio de um argumento String, um método de fábrica ou uma classe de implementação específica. Esses algoritmos têm fraquezas criptográficas fatais, que tornam trivial quebrar em um prazo razoável. Algoritmos fortes devem resistir a ataques muito além do possível.

Exemplo da vulnerabilidade Use Of Deprecated Or Obsolete Functions (Uso de funções depreciadas ou obsoletas):

Na imagem a seguir na criarHashSenha, a aplicação protege dados sensíveis usando um algoritmo criptográfico, digest, que é considerado fraco ou até mesmo trivialmente quebrado, no trecho em vermelho.

```
function criarHashSenha(senha, salt) {  
  const senhaSalt = senha + salt;  
  const hashSenha = crypto.createHash('sha256').update(senhaSalt).digest('hex');  
  return hashSenha;  
}
```

Recomendações gerais:

Como evitá-lo?

- Use apenas algoritmos criptográficos fortes e aprovados, incluindo AES, RSA, ECC e SHA-256, respectivamente, entre outros.
- Não use algoritmos fracos que são considerados completamente quebrados, como DES, RC4 e MD5, entre outros.
- Evite, sempre que possível, usar algoritmos legados que não são considerados "à prova de futuro" com margens de segurança suficientes, mesmo que sejam considerados "seguros o suficiente" para hoje. Isso inclui algoritmos que são mais fracos do que deveriam e têm substituições mais fortes, mesmo que ainda não estejam fatalmente quebrados - como SHA-1, 3DES,
- Considere o uso de um conjunto oficial relevante de classificações, como NIST ou ENISA. Se possível, use apenas implementações de algoritmo certificadas FIPS 140-2.

Vulnerabilidade Use Of Hardcoded Password (Uso de senha codificada) -->

Como surge Use Of Hardcoded Password (Uso de senha codificada)?

A base de código do aplicativo tem senhas literais de cadeia de caracteres incorporadas no código-fonte. Esse valor codificado é usado para comparar com credenciais fornecidas pelo usuário ou para autenticar downstream em um sistema remoto (como um banco de dados ou um serviço Web remoto). Um invasor só precisa obter acesso ao código-fonte para revelar a senha codificada. Da mesma forma, o invasor pode fazer engenharia reversa dos binários do aplicativo compilado e recuperar facilmente a senha incorporada.

Exemplo da vulnerabilidade Use Of Hardcoded Password (Uso de senha codificada):

Na imagem a seguir, a aplicação usa a senha codificada "%" para fins de autenticação, seja para verificar a identidade dos usuários ou para acessar outro sistema remoto. Essa senha no trecho em vermelho, aparece no código, o que implica que está acessível a qualquer pessoa com acesso ao código-fonte e não pode ser alterado sem reconstruir a aplicação.

```
function percentEncode(c) {  
  let hex = c.toString(16).toUpperCase();  
  if (hex.length === 1) {  
    hex = "0" + hex;  
  }  
  
  return "%" + hex;  
}
```

Recomendações gerais:

Como evitá-lo?

- Não codifique nenhum dado secreto no código-fonte, especialmente senhas.
- Em particular, as senhas de usuário devem ser armazenadas em um banco de dados ou serviço de diretório e protegidas com um hash de senha forte (por exemplo, bcrypt, scrypt, PBKDF2 ou Argon2). Não compare senhas de usuário com um valor codificado.
- As senhas do sistema devem ser armazenadas em um arquivo de configuração ou no banco de dados e protegidas com criptografia forte (por exemplo, AES-256). As chaves de criptografia devem ser gerenciadas com segurança e não codificadas.

Vulnerabilidade Use Of HTTP Sensitive Data Exposure (Uso de exposição de dados confidenciais HTTP) -->

Como surge Use Of HTTP Sensitive Data Exposure (Uso de exposição de dados confidenciais HTTP)?

O servidor web criado usa HTTP sem qualquer criptografia TLS.

Exemplo da vulnerabilidade Use Of HTTP Sensitive Data Exposure (Uso de exposição de dados confidenciais HTTP):

Na imagem a seguir, o servidor web criado em `createServer` no trecho em vermelho, utiliza HTTP, o que implica que o todo o tráfego para e deste servidor não é criptografado.

```
app.listen = function listen() {  
  var server = http.createServer(this);  
  return server.listen.apply(server, arguments);  
};
```

Recomendações gerais:

Como evitá-lo?

- Nunca transmita informações confidenciais em texto simples
- Certifique-se sempre de que as conexões, especialmente em redes não confiáveis, estejam protegidas por TLS bem configurado

Vulnerabilidade Unsafe Use Of Target Blank (Uso não seguro do destino em branco) -->

Como surge Unsafe Use Of Target Blank (Uso não seguro do destino em branco)?

Ao abrir uma nova página usando um elemento HTML `<a>` com o atributo "target" (com qualquer valor) ou com `window.open()` dentro do JavaScript, a nova página tem algum acesso à página original por meio do objeto `window.opener`. Isso pode permitir o redirecionamento para uma página de phishing mal-intencionada.

Exemplo da vulnerabilidade Unsafe Use Of Target Blank (Uso não seguro do destino em branco):

Na imagem a seguir, usar `` no trecho em vermelho, sem definir corretamente o atributo `rel` ou desassociar a nova janela de seu pai, é uma maneira insegura de abrir uma nova janela.

```
function buscarVulnerabilidades(nome) {
  // Verifique se o nome não está vazio
  if (!nome.trim()) {
    console.log('O campo de busca está vazio.');
```

`return;`

```
  }

  fetch(`http://localhost:3000/vulnerabilidades?nome=${nome}`)
    .then(res => res.json())
    .then((json) => {
      console.log(json);

      const cards = document.getElementById('cards');
      cards.innerHTML = "";

      if (json.success) {
        json.userInfo.forEach((item) => {
          const card = document.createElement("div");
          card.classList.add("card");

          const cardContent = `
            <div href="http://localhost:3000/vulnerabilidades/${item.id}">
              <span class="item-name">${item.nome}</span>
              <p>${item.descricao}</p>
              <p class="item-referencia"> Referência:
                <a href="${item.link}" target="_blank" class="item-link"> ${item.link}</a>
              </p>
            </div>
          `;

          cardContent.innerHTML = cardContent;
          card.innerHTML = cardContent;
          cards.appendChild(card);
        });
      } else {
        console.log('Dados da tabela de vulnerabilidades não encontrados');

        const mensagem = document.createElement("p");
        mensagem.innerText = "Nenhum dado encontrado na tabela de vulnerabilidades.";
        cards.appendChild(mensagem);

        cards.innerHTML = `${input.value}`;
      }
    })
    .catch(error => console.error('Erro na requisição:', error));
}
```

Recomendações gerais:

Como evitá-lo?

Para HTML:

- Não defina o atributo "target" (com qualquer valor) para links criados por usuários, a menos que necessário.

Se necessário, ao usar o atributo "target", defina também o atributo "rel" como "noopener noreferrer":

- "noopener" para Chrome e Opera
- "noreferrer" para Firefox e navegadores antigos
- Nenhuma solução semelhante para o Safari

Para JavaScript:

- Ao invocar uma nova janela não confiável usando "var newWindow = window.open()", defina "newWindow.opener=null" antes de definir "newWindow.location" para um site potencialmente não confiável, de modo que, quando o novo site é aberto na nova janela, ele não tem acesso ao seu atributo "opener" original

Vulnerabilidade Unprotected Cookie (Cookie desprotegido) -->

Como surge Unprotected Cookie (Cookie desprotegido)?

A estrutura do aplicativo Web, por padrão, não define o sinalizador "httpOnly" para o cookie sessionid do aplicativo e outros cookies de aplicativos confidenciais. Da mesma forma, o aplicativo não usa explicitamente o sinalizador de cookie "httpOnly", permitindo que scripts de cliente acessem os cookies por padrão. Da mesma forma, os cookies que não possuem o sinalizador "seguro" serão enviados automaticamente pelo navegador para o servidor web, independentemente da segurança do protocolo subjacente, como HTTP desprotegido. Os cookies sinalizados com o atributo "secure" só serão enviados através de uma conexão HTTPS segura.

Exemplo da vulnerabilidade Unprotected Cookie (Cookie desprotegido):

Na imagem a seguir, o método getSessionID da aplicação web cria um nome de cookie, no trecho em vermelho, e o retorna na resposta. No entanto, a aplicação não está configurada para definir automaticamente o cookie com o atributo httpOnly, e o código não adiciona explicitamente isso ao cookie.

```
function getSessionID() {  
    const cookies = document.cookie.split(';');  
    for (const cookie of cookies) {  
        const [name, value] = cookie.trim().split('=');  
        if (name === 'sessionID') {  
            return value;  
        }  
    }  
    return null;  
}
```

Recomendações gerais:

Como evitá-lo?

- Sempre defina o sinalizador "httpOnly" para qualquer cookie sensível do lado do servidor.
- É altamente recomendável implementar HTTP Strict Transport Security (HSTS) para garantir que o cookie será enviado através de um canal seguro. - Defina explicitamente o sinalizador "httpOnly" para cada cookie definido pelo aplicativo. - Configure e defina todos os cookies sensíveis a serem criados com o atributo "secure".

Vulnerabilidade Potentially Vulnerable To CSRF (Potencialmente vulnerável à CSRF) -->

Como surge Potentially Vulnerable To CSRF (Potencialmente vulnerável à CSRF)?

O aplicativo executa alguma ação que modifica o conteúdo do banco de dados, com base puramente no conteúdo da solicitação HTTP, e não requer autenticação renovada por solicitação (como autenticação de transação ou um token sincronizador), em vez disso, depende apenas da autenticação de sessão. Isso significa que um invasor pode usar engenharia social para fazer com que uma vítima navegue até um link que contém uma solicitação de transação para o aplicativo vulnerável, enviando essa solicitação do navegador do usuário. Uma vez que o aplicativo recebe a solicitação, ele confiaria na sessão da vítima e executaria a ação.

Exemplo da vulnerabilidade Potentially Vulnerable To CSRF (Potencialmente vulnerável à CSRF):

Na imagem a seguir, o método gindefinido no trecho em vermelho, recebe um parâmetro de uma solicitação do usuário do aplicativo. O valor desse parâmetro flui pelo código e é eventualmente usado para acessar a funcionalidade de alteração do estado do aplicativo. Isso pode permitir o Cross-Site request Forgery (CSRF).

```
const app = express();
```

Recomendações gerais:

Como evitá-lo?

- Mitigar o CSRF requer uma camada adicional de autenticação incorporada ao mecanismo de validação de solicitação, esse mecanismo anexaria um token adicional que se aplica apenas ao usuário determinado; este token estaria disponível na página web do usuário, mas não será anexado automaticamente a uma solicitação de um site diferente (por exemplo, não armazenado em um cookie), Uma vez que o token não é anexado automaticamente à solicitação, e não está disponível para o invasor, e é exigido pelo servidor para processar a solicitação, seria completamente impossível para o invasor preencher um formulário válido entre sites que contenha esse token, Muitas plataformas oferecem funcionalidade de mitigação de CSRF integrada que deve ser usada, e executar esse tipo de gerenciamento de token sob o capô, alternativamente, use uma biblioteca conhecida ou confiável que adicione essa funcionalidade, Se a implementação da proteção CSRF for necessária, essa proteção deve seguir as seguintes regras: Qualquer formulário de alteração de estado (operações Criar, Atualizar, Excluir) deve impor a proteção CSRF, adicionando um token CSRF a cada estado alterando o envio de formulário no cliente, Um token CSRF deve ser gerado e ser exclusivo por usuário por sessão (e, de preferência, por solicitação), O token CSRF deve ser inserido no formulário do lado do cliente e ser enviado ao servidor como parte da solicitação de formulário, Por exemplo, pode ser um campo oculto em um formulário HTML ou um cabeçalho personalizado adicionado por uma solicitação Javascript, O token CSRF no corpo da solicitação ou no cabeçalho personalizado deve ser verificado como pertencente ao usuário atual pelo servidor, antes que uma solicitação seja autorizada e processada como válida, Sempre confie nas práticas recomendadas ao usar a proteção XSRF - sempre habilite a funcionalidade interna ou as bibliotecas disponíveis sempre que possível.

- Ao usar a proteção XSRF em todo o aplicativo, nunca desabilite ou subverta explicitamente a proteção XSRF para uma funcionalidade específica, a menos que essa funcionalidade tenha sido completamente verificada para não exigir proteção XSRF.

Vulnerabilidade Potential Clickjacking On Legacy Browsers (Potencial Clickjacking em navegadores legados) -->

Como surge Potential Clickjacking On Legacy Browsers (Potencial Clickjacking em navegadores legados)?

A causa raiz da vulnerabilidade a um ataque de clickjacking é que as páginas da Web do aplicativo podem ser carregadas em um quadro de outro site. O aplicativo não implementa um script de quebra de quadros adequado, o que impediria que a página fosse carregada em outro quadro. Observe que existem muitos tipos de scripts de redirecionamento simplistas que ainda deixam o aplicativo vulnerável a técnicas de clickjacking e não devem ser usados. Ao lidar com navegadores modernos, os aplicativos atenuam essa vulnerabilidade emitindo cabeçalhos apropriados Content-Security-Policy ou X-Frame-Options para indicar ao navegador para não permitir o enquadramento. No entanto, muitos navegadores herdados não oferecem suporte a esse recurso e exigem uma abordagem mais manual, implementando uma mitigação em Javascript. Para garantir o suporte legado, um script de framebusting é necessário.

Exemplo da vulnerabilidade Potential Clickjacking On Legacy Browsers (Potencial Clickjacking em navegadores legados):

Na imagem a seguir, o aplicativo não protege a página da Web html contra ataques de clickjacking em navegadores legados, usando scripts framebusting.

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="../../gustavo_projeto_webgoat_web_aplicacion/css/criar_dados_vulnerabilidades.css">
  <link rel="stylesheet" href="../../gustavo_projeto_webgoat_web_aplicacion/css/enviar_arquivo.css">
  <script src="../../gustavo_projeto_webgoat_web_aplicacion/js/criar_dados_vulnerabilidades.js"></script>
  <script src="../../gustavo_projeto_webgoat_web_aplicacion/js/salvar_dados_vulnerabilidades_em_xml.js"></script>
  <title>Inserir Vulnerabilidade</title>
</head>
<body>
  <div class="div_principal_criar_vulnerabilidades">
    <div class="div_criar_dados_vulnerabilidades">
      <h1 class="text_inserir_vulnerabilidade">Inserir Vulnerabilidade
      <button class="button_ir_para_filtrar_vulnerabilidades" type="button" onclick="irParaFiltrar()">Ir até a filtragem</button>
      </h1>

      <form id="vulnerabilidadeForm">
        <div class="container_inserir_nome_vulnerabilidade">
          <label class="text_nome_vulnerabilidade" for="text_nome_vulnerabilidade">Nome da vulnerabilidade:</label>
          <input type="text" id="input_nome_vulnerabilidade" name="nome" required>
        </div>

        <div class="container_inserir_descricao_vulnerabilidade">
          <label class="text_descricao_vulnerabilidade" for="text_descricao_vulnerabilidade">Descrição da vulnerabilidade:</label>
          <textarea id="input_descricao_vulnerabilidade" name="descricao" required></textarea>
        </div>

        <div class="container_inserir_link_vulnerabilidade">
          <label class="text_link_vulnerabilidade" for="text_link_vulnerabilidade">Link/referência da vulnerabilidade:</label>
          <input type="text" id="input_link_vulnerabilidade" name="link" required>
        </div>

        <div class="div_botoes_inserir_vulnerabilidades">
          <button class="button_inserir_vulnerabilidade" type="button" onclick="inserirVulnerabilidade()">Inserir vulnerabilidades</button>
          <button class="button_inserir_vulnerabilidade_em_xss" type="button" onclick="salvarEmXMLEEnviarParaBanco()">Inserir vulnerabilidades em XML</button>
        </div>
      </form>
    </div>
    <div class="div_enviar_arquivo">
      <label class="text_selecao_arquivo" for="arquivo">Selecionar arquivo XML</label>
      <input type="file" id="arquivo">
      <br><br>
      <textarea id="preview" cols="50" rows="8"></textarea>
      <br><br>
      <div class="div_botoes_download_enviar_banco_dados">
        <button id="enviarParaBanco">Enviar para Banco de Dados</button>
        <button id="download">Baixar arquivo</button>
      </div>
    </div>
  </div>
  <script src="../../gustavo_projeto_webgoat_web_aplicacion/js/file.js"></script>

  <script>
    function irParaFiltrar() {
      window.location.href = 'filtrar_vulnerabilidades.html';
    }
  </script>
</body>
</html>

```

Recomendações gerais:

Como evitá-lo?

Orientações genéricas:

- Defina e implemente uma Política de Segurança de Conteúdo (CSP) no lado do servidor, incluindo uma diretiva de ancestrais de quadro. Aplique o CSP em todas as páginas da Web relevantes.
- Se determinadas páginas da Web precisarem ser carregadas em um quadro, defina uma URL de destino específica na lista de permissões.
- Como alternativa, retorne um cabeçalho "X-Frame-Options" em todas as respostas HTTP. Se for necessário permitir que uma determinada página da Web seja carregada em um quadro, defina uma URL de destino específica na lista de permissões.

- Para suporte legado, implemente código framebusting usando Javascript e CSS para garantir que, se uma página for enquadrada, ela nunca seja exibida e tente navegar até o quadro para evitar ataques. Mesmo que a navegação falhe, a página não é exibida e, portanto, não é interativa, mitigando possíveis ataques de clickjacking.
-

Recomendações especificações:

- Implemente um script de framebuster adequado no cliente, que não seja vulnerável a ataques de quebra de frame.
- O código deve primeiro desabilitar a interface do usuário, de modo que, mesmo que a quebra de quadros seja evitada com êxito, a interface do usuário não possa ser clicada. Isso pode ser feito definindo o valor CSS do atributo "display" como "none" nas tags "body" ou "html". Isso é feito porque, se um quadro tentar redirecionar e se tornar o pai, o pai mal-intencionado ainda pode impedir o redirecionamento por meio de várias técnicas.
- O código deve então determinar se nenhum enquadramento ocorre comparando `auto === topo`; se o resultado for true, a interface do usuário pode ser habilitada. Se for false, tente navegar para fora da página de enquadramento definindo o atributo `top.location` como `self.location`.

Vulnerabilidade Open Redirect (Abrir Redirecionamento) -->

Como surge Open Redirect (Abrir Redirecionamento)?

O aplicativo redireciona o navegador do usuário para uma URL fornecida por uma entrada contaminada, sem primeiro garantir que a URL leva a um destino confiável e sem avisar os usuários de que eles estão sendo redirecionados para fora do site atual. Um invasor pode usar engenharia social para fazer com que uma vítima clique em um link para o aplicativo com um parâmetro definindo outro site para o qual o aplicativo redirecionará o navegador do usuário. Uma vez que o usuário pode não estar ciente do redirecionamento, ele pode estar sob o equívoco de que o site que ele está navegando atualmente pode ser confiável.

Exemplo da vulnerabilidade Open Redirect (Abrir Redirecionamento):

Na imagem a seguir, o potencialmente valor contaminado fornecido pelo nome no trecho em amarelo, é usado como um URL de destino por json no trecho em vermelho, permitindo potencialmente que atacantes realizem uma refireção aberta.

```

app.get('/vulnerabilidades/novafiltragem/:nome', (request, response) => {
  let nome = request.params.nome;

  const query = `SELECT * FROM tbl_vulnerabilidades WHERE nome LIKE '%${nome}%'`;

  db.query(query, (err, result) => {
    if (err) {
      console.error('Erro ao tentar retornar os dados:', err);
      return response.status(500).json({ success: false, message: 'Erro na busca pelos dados na tabela de vulnerabilidades' });
    }

    if (result.length > 0) {
      console.log('Dados de tabela de vulnerabilidades retornados!');
      return response.json({ success: true, userInfo: result });
    }
  });

  response.json(query)
});

```

Recomendações gerais:

Como evitá-lo?

- O ideal é não permitir URLs arbitrárias para redirecionamento. Em vez disso, crie um mapeamento de valores de parâmetro fornecidos pelo usuário para URLs legítimos.
- Se for necessário permitir URLs arbitrárias:

Para URLs dentro do site do aplicativo, primeiro filtre e codifique o parâmetro fornecido pelo usuário e, em seguida:

- Criar uma lista branca de URLs permitidos dentro do aplicativo
- Use variáveis como uma URL relativa como uma URL absoluta, prefixando-a com o domínio do site do aplicativo - isso garantirá que todo o redirecionamento ocorra dentro do domínio

Redirecionamento de lista branca para domínios externos permitidos filtrando primeiro URLs com prefixos confiáveis, Os prefixos devem ser testados até a terceira barra \[/\] - 'scheme://my,trusted,domain,com/', para evitar evasão, Por exemplo, se a terceira barra \[/\] não for validada e scheme://my,trusted,domain,com for confiável, o esquema de URL://my,trusted,domain,com,evildomain,com seria válido sob este filtro, mas o domínio que realmente está sendo navegado é evildomain,com, não domain,com.

- Para redirecionamento aberto totalmente dinâmico, use uma página de isenção de responsabilidade intermediária para fornecer aos usuários um aviso claro de que eles estão deixando o site.

Vulnerabilidade Log Forging (Forjamento de log) -->

Como surge Log Forging (Forjamento de log)?

O aplicativo grava logs de auditoria em ações sensíveis à segurança. Como o log de auditoria inclui a entrada do usuário que não é verificada quanto à validade do tipo de dados nem posteriormente limpa, a entrada pode conter informações falsas feitas para se parecerem com dados legítimos do log de auditoria.

Exemplo da vulnerabilidade Log Forging (Forjamento de log):

Na imagem a seguir, o método Lambda no trecho em amarelo, obtém a entrada do usuário do elemento body. O valor deste elemento flui pelo código sem ser devidamente sanitizado ou validado e, eventualmente, é usado na escrita de um log de auditoria no Lambda no trecho em vermelho. Isso pode permitir a falsificação de logs.

```
app.post('/processar-xml', (req, res) => {  
  const xml = req.body;  
  
  try {  
    const xmlDoc = libxmljs.parseXmlString(xml, {  
      noent: false, // Habilitar o processamento de entidades externas  
      // Outras opções, se necessário  
    });  
  
    console.log(xmlDoc.toString());  
    res.status(200).json({ success: true, message: 'XML processado com sucesso!' });  
  } catch (error) {  
    console.error('Erro ao analisar XML:', error);  
    res.status(500).send('Erro ao processar XML');  
  }  
});
```

Recomendações gerais:

Como evitá-lo?

- Validar todas as entradas, independentemente da fonte, A validação deve ser baseada em uma lista branca: aceite apenas dados que se ajustem a uma estrutura especificada, em vez de rejeitar padrões incorretos, Verificar para: Tipo de dado Tamanho Gama Formato Valores esperados 3, A validação não substitui a

codificação, Codificar totalmente todos os dados dinâmicos, independentemente da origem, antes de incorporá-los em logs, 4, Use um mecanismo de registro seguro.

Vulnerabilidade JSON Hijacking (Sequestro JSON) -->

Como surge JSON Hijacking (Sequestro JSON)?

Para que o aplicativo seja vulnerável, é necessário o seguinte: * O aplicativo deve estar usando autenticação baseada em cookie * O aplicativo responde de forma autenticada à simples solicitação GET * Esses dados confidenciais são retornados como um JSON, especificamente em forma de matriz (entre colchetes quadrados `[]`, e contendo objetos dentro da matriz Ao retornar uma matriz JSON, um invasor pode criar um site mal-intencionado, que incorpora uma marca `<script>` em sua página da seguinte maneira: `'<script src="https://example.com/path/to/vulnerable/page"></script>'` O navegador interpretará o valor retornado como um objeto, fazendo com que ele exista temporariamente no DOM da página da Web maliciosa; no entanto, como esse objeto não é atribuído ou referenciado, ele será efêmero, e normalmente seria imediatamente descartado. Isso é semelhante a qualquer outra declaração ou valor de retorno sem uma atribuição, e a página da Web maliciosa seria incapaz de fazer referência a ela de forma alguma.

Exemplo da vulnerabilidade JSON Hijacking (Sequestro JSON):

Na imagem a seguir, o aplicativo retorna dados confidenciais em um objeto JSON no trecho em vermelho.

```
function register() {
  const login = document.getElementById('registerLogin').value;
  const nome = document.getElementById('registerName').value;
  const senha = document.getElementById('registerPassword').value;

  fetch('http://localhost:3000/register', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ login, nome, senha }),
  })
  .then(response => response.json())
  .then(data => {
    if (data.success) {
      console.log("User registered");
    } else {
      console.error("Registration failed:", data.message);
    }
  })
  .catch(error => {
    console.error("Error:", error);

    const errorMessage = document.createElement("p");
    errorMessage.innerHTML = `Error: ${error}`;
    document.body.appendChild(errorMessage);
  });
}
```


Recomendações gerais:

Como evitá-lo?

Há vários métodos para resolver esse problema:

- Não responda com matrizes JSON, pois elas são encapsuladas com colchetes, que são imediatamente avaliados como objetos; Se necessário, envolva a matriz com um objeto externo (por exemplo.
- {"matriz":\[\]}) ou adicione algum tipo de prefixo para evitar esse problema
- Se necessário, responda com matrizes JSON somente à solicitação POST; garantir que nenhuma informação confidencial seja retornada como uma matriz para uma solicitação GET
- Prefixe o objeto JSON com JavaScript ('for(;;); ') ou um JSON não analisável ('{};') e, antes de processá-lo no cliente, remova esse prefixo; o último fará com que a importação falhe, e o primeiro fará com que a importação trave para sempre - de qualquer forma, isso impedirá que um invasor importe um objeto JSON como um script.

Vulnerabilidade Information Exposure Through An Error Message (Exposição de informações por meio de uma mensagem de erro) -->

Como surge Information Exposure Through An Error Message (Exposição de informações por meio de uma mensagem de erro)?

O aplicativo manipula exceções de maneira insegura, incluindo detalhes brutos diretamente na mensagem de erro. Isso pode ocorrer de várias maneiras: não manipulando a exceção; imprimi-lo diretamente na saída ou arquivo; retornar explicitamente o objeto de exceção; ou por configuração. Esses detalhes de exceção podem incluir informações confidenciais que podem vaziar para os usuários devido à ocorrência do erro de tempo de execução.

Exemplo da vulnerabilidade Information Exposure Through An Error Message (Exposição de informações por meio de uma mensagem de erro):

Na imagem a seguir, o método Lambda, no trecho em amarelo, trata um erro de exceção ou tempo de execução. Durante o código de tratamento de exceção, a aplicação expõe os detalhes da exceção em json, no método Lambda no trecho em vermelho.

```
app.post('/xml-data', (req, res) => {
  try {
    const xmlObj = req.body.xml;

    if (!xmlObj || typeof xmlObj !== 'string') {
      return res.status(400).json({ success: false, message: 'O parâmetro "xml" é obrigatório e deve ser uma string para a inserção.' });
    }

    try {
      const xmlDoc = libxmljs.parseXml(xmlObj.toString(), {
        noent: true, // Habilitar o processamento de entidades externas
        // Outras opções, se necessário
      });

      console.log(xmlDoc.toString());
      const timestamp = new Date().getTime();
      const randomPart = Math.floor(Math.random() * 1000);
      const fileName = `arquivo_${timestamp}_${randomPart}.xml`;

      const xmlString = xmlDoc.toString();

      fs.writeFile(fileName, xmlString, (err) => {
        if (err) {
          console.error('Erro ao escrever o arquivo:', err);
          return res.status(500).json({ success: false, message: 'Erro ao escrever o arquivo' });
        }

        res.status(200).json({ success: true, fileName });
      });
    } catch (error) {
      console.error('Erro ao analisar XML:', error);
      res.status(500).json({ success: false, message: 'Erro ao processar XML' });
    }
  } catch (error) {
    console.error('Erro no servidor:', error);
    res.status(500).json({ success: false, message: 'Erro interno no servidor', error: error.message });
  }
});
```

Recomendações gerais:

Como evitá-lo?

- Não exponha dados de exceção diretamente à saída ou aos usuários, em vez disso, retorne uma mensagem de erro genérica e informativa. Registre os detalhes da exceção em um mecanismo de log dedicado.

Qualquer método que possa lançar uma exceção deve ser encapsulado em um bloco de manipulação de exceção que:

- Manipula explicitamente as exceções esperadas.
- Inclui uma solução padrão para lidar explicitamente com exceções inesperadas.
- Configure um manipulador global para evitar que erros não tratados saiam do aplicativo.

Vulnerabilidade Client Weak Cryptographic Hash (Hash criptográfico fraco do cliente) -->

Como surge Client Weak Cryptographic Hash (Hash criptográfico fraco do cliente)?

Um método fraco de hash é usado no código do cliente.

Exemplo da vulnerabilidade Client Weak Cryptographic Hash (Hash criptográfico fraco do cliente):

Na imagem a seguir, o aplicativo emprega hash fraco em sha1 no trecho em vermelho.

```
exports.unsign = function(val, secret){
  if ('string' !== typeof val) throw new TypeError("Signed cookie string must be provided.");
  if ('string' !== typeof secret) throw new TypeError("Secret string must be provided.");
  var str = val.slice(0, val.lastIndexOf('.'))
  , mac = exports.sign(str, secret);

  return sha1(mac) == sha1(val) ? str : false;
};
```

Recomendações gerais:

Como evitá-lo?

- O hash de uma senha no cliente, em vez do servidor, torna a autenticação possível usando o próprio hash em vez de uma senha; isso significa que uma senha com hash é tão boa quanto uma senha normal, pois o hash não ocorre no servidor - deixando o servidor vulnerável a ataques do tipo Pass-the-Hash, onde a autenticação é feita com o hash em vez de credenciais de login. Para fins de autenticação - evitando hashing no cliente.

Sempre que possível - evite usar criptografia no client:

- A criptografia deve ser feita no servidor
- Garantir a confidencialidade e a integridade dos dados em trânsito deve estar à altura da implementação adequada e segura do TLS
- Dados confidenciais nunca devem ser armazenados no navegador, mesmo que criptografados

Se necessário, identifique uma forma segura de hashing e manuseio:

- Usar algoritmos de hash criptograficamente seguros, como SHA512
- Derive saís, em vez de reutilizar o mesmo sal em várias instâncias de hash

Vulnerabilidade Client Server Empty Password (Senha vazia do servidor cliente) -->

Como surge Client Server Empty Password (Senha vazia do servidor cliente)?

A senha definida está vazia, indicando que ela está codificada no aplicativo e pode ser facilmente adivinhada por qualquer pessoa que tente acessar a conta à qual essa senha pertence.

Exemplo da vulnerabilidade Client Server Empty Password (Senha vazia do servidor cliente):

Na imagem a seguir, a aplicação usa a senha vazia password para fins de autenticação, seja para verificar a identidade dos usuários ou para acessar outro sistema remoto. Essa senha vazia é definida no trecho em vermelho, no código não pode ser alterada sem reconstruir a aplicação e indica que a conta associada está exposta.

```
module.exports.setThePassword = function (url, password) {  
  url.password = "";  
  const decoded = punycode.ucs2.decode(password);  
  for (let i = 0; i < decoded.length; ++i) {  
    url.password += percentEncodeChar(decoded[i], isUserInfoPercentEncode);  
  }  
};
```

Recomendações gerais:

Como evitá-lo?

- Não permitir o uso de senha vazia para quaisquer contas de acesso a sistemas, serviços ou informações, tanto por política quanto pelos meios técnicos disponíveis.
- Não codifique nenhum dado secreto no código-fonte, especialmente senhas.
- Em particular, as senhas de usuário devem ser armazenadas em um banco de dados ou serviço de diretório e protegidas com um hash de senha forte (por exemplo, bcrypt, scrypt, PBKDF2 ou Argon2). Não compare senhas de usuário com um valor codificado.
- As senhas do sistema devem ser armazenadas em um arquivo de configuração ou no banco de dados e protegidas com criptografia forte (por exemplo, AES-256). As chaves de criptografia devem ser gerenciadas com segurança e não codificadas.
-

Vulnerabilidade Client Potential DOM Open Redirect (Redirecionamento aberto do DOM em potencial do cliente) -->

Como surge Client Potential DOM Open Redirect (Redirecionamento aberto do DOM em potencial do cliente)?

O aplicativo redireciona o navegador do usuário para uma URL fornecida por uma entrada contaminada, sem primeiro garantir que a URL leva a um destino confiável e sem avisar os usuários de que eles estão sendo redirecionados para fora do site atual. Um invasor pode usar engenharia social para fazer com que uma vítima clique em um link para o aplicativo com um parâmetro definindo outro site para o qual o aplicativo redirecionará o navegador do usuário. Uma vez que o usuário pode não estar ciente do redirecionamento, ele pode estar sob o equívoco de que o site que ele está navegando atualmente pode ser confiável.

Exemplo da vulnerabilidade Client Potential DOM Open Redirect (Redirecionamento aberto do DOM em potencial do cliente):

Na imagem a seguir o valor potencialmente contaminado fornecido pelo valor no trecho em vermelho, é usado como URL de destino por href no trecho em amarelo, permitindo potencialmente que atacantes realizem uma redireção aberta.

```
const download = function(){
  const a = document.createElement('a');
  a.style = 'display: none';
  document.body.appendChild(a);

  return function(conteudo, nomeArquivo){
    const blob = new Blob([conteudo], { type: 'octet/stream' });

    console.log("resultado do conteudo e do arquivo" + conteudo + nomeArquivo)

    console.log("resultado somente do conteudo" + conteudo)

    console.log("resultado somente do nome do arquivo" + nomeArquivo)

    const url = window.URL.createObjectURL(blob);
    a.href = url;
    a.download = nomeArquivo;
    a.click();
    window.URL.revokeObjectURL(url);

    console.log("resultado da URL" + url)

  }
}

btnDownload.addEventListener('click', function(){
  download()(preview.value, fileName);
  console.log("resultado do DOWNLOAD " + preview.value)
})
```

Recomendações gerais:

Como evitá-lo?

- O ideal é não permitir URLs arbitrárias para redirecionamento. Em vez disso, crie um mapeamento de valores de parâmetro fornecidos pelo usuário para URLs legítimos.
- Se for necessário permitir URLs arbitrárias:

Para URLs dentro do site do aplicativo, primeiro filtre e codifique o parâmetro fornecido pelo usuário e, em seguida:

- Criar uma lista branca de URLs permitidos dentro do aplicativo
- Use variáveis como uma URL relativa como uma URL absoluta, prefixando-a com o domínio do site do aplicativo - isso garantirá que todo o redirecionamento ocorra dentro do domínio

Para URLs fora do aplicativo (se necessário):

Redirecionamento de lista branca para domínios externos permitidos filtrando primeiro URLs com prefixos confiáveis. Os prefixos devem ser testados até a terceira barra `[/\]` - 'scheme://my,trusted,domain,com/', para evitar evasão. Por exemplo, se a terceira barra `[/\]` não for validada e 'scheme://my,trusted,domain,com' for confiável, o esquema de URL: `//my,trusted,domain,com,evildomain,com` seria válido sob este filtro, mas o domínio que realmente está sendo navegado é `evildomain,com`, não `domain,com`.

- Para redirecionamento aberto totalmente dinâmico, use uma página de isenção de responsabilidade intermediária para fornecer aos usuários um aviso claro de que eles estão deixando o site.

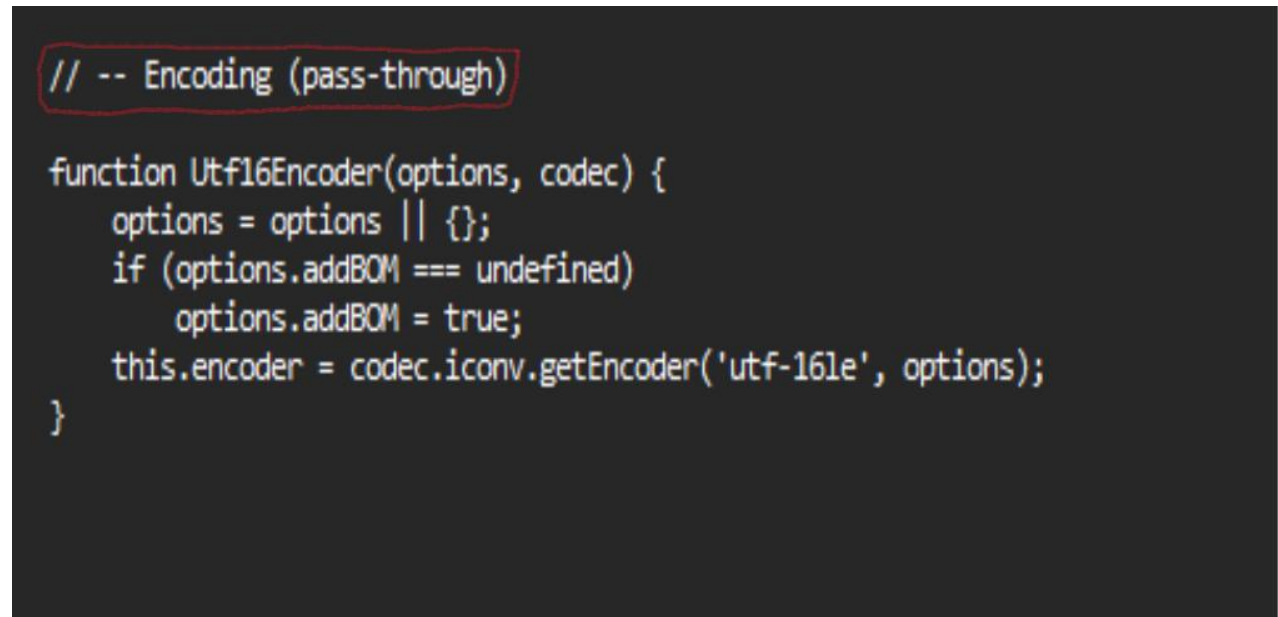
Vulnerabilidade Client Password In Comment (Senha do cliente em comentário) -->

Como surge Client Password In Comment (Senha do cliente em comentário)?

Um aplicativo bem desenvolvido terá seu código-fonte bem comentado. Muitas vezes, os programadores deixam informações de implantação em comentários ou retêm dados de depuração que foram usados durante o desenvolvimento. Esses comentários geralmente contêm dados secretos, como senhas. Esses comentários de senha são armazenados no código-fonte perpetuamente e não são protegidos.

Exemplo da vulnerabilidade Client Password In Comment (Senha do cliente em comentário):

Na imagem a seguir a aplicação contém senhas incorporadas em comentários de código-fonte, como pass- no trecho em vermelho, que podem ser facilmente visualizadas pelos usuários.

A screenshot of a code editor with a dark background. The first line of code is a comment: `// -- Encoding (pass-through)`. This line is highlighted with a red rectangular box. Below the comment is a JavaScript function definition: `function Utf16Encoder(options, codec) {`, followed by several lines of code: `options = options || {};`, `if (options.addBOM === undefined)`, `options.addBOM = true;`, `this.encoder = codec.iconv.getEncoder('utf-16le', options);`, and finally `}`.

```
// -- Encoding (pass-through)

function Utf16Encoder(options, codec) {
  options = options || {};
  if (options.addBOM === undefined)
    options.addBOM = true;
  this.encoder = codec.iconv.getEncoder('utf-16le', options);
}
```

Recomendações gerais:

Como evitá-lo?

- Não armazene segredos, como senhas, em comentários de código-fonte.

Vulnerabilidade Client Insecure Randomness (Aleatoriedade insegura do cliente) -->

Como surge Client Insecure Randomness (Aleatoriedade insegura do cliente)?

O aplicativo usa um método fraco de gerar valores pseudoaleatórios, de modo que outros números poderiam ser determinados a partir de um tamanho de amostra relativamente pequeno. Uma vez que o gerador de números pseudoaleatórios usado é projetado para distribuição estatisticamente uniforme de valores, ele é aproximadamente determinístico. Assim, depois de coletar alguns valores gerados, seria possível para um invasor calcular valores passados ou futuros. Especificamente, se esse valor pseudoaleatório for usado em qualquer contexto de segurança, como senhas de uso único, chaves, identificadores secretos ou saís - um invasor provavelmente seria capaz de prever o próximo valor gerado e roubá-lo, ou adivinhar um valor gerado anteriormente e falsificar sua intenção original.

Exemplo da vulnerabilidade Client Insecure Randomness (Aleatoriedade insegura do cliente):

Na imagem a seguir, o método generateSessionID no trecho em vermelho usa um método fraco de aleatoriedade para produzir valores aleatórios. Esses valores podem ser usados como identificadores pessoais, tokens de sessão ou entrada criptográfica; no entanto, devido à sua aleatoriedade insuficiente, um atacante pode ser capaz de derivar seu valor.

```
function generateSessionID(userID) {  
  // Use o userID para tornar a geração mais única  
  const randomPart = Math.floor(Math.random() * 1000000);  
  return `${userID}-${randomPart}`;  
}
```

Recomendações gerais:

Como evitá-lo?

- Sempre use um gerador de números pseudoaleatórios criptograficamente seguro, em vez de métodos aleatórios básicos, particularmente ao lidar com um contexto de segurança
- Use o gerador cryptorandom integrado à sua linguagem ou plataforma e certifique-se de que ele seja semeado com segurança. Não semeie o gerador com uma semente fraca e não aleatória. (Na maioria dos casos, o padrão é seguramente aleatório).
- Certifique-se de usar um valor aleatório longo o suficiente, tornando os ataques de força bruta inviáveis.

Vulnerabilidade Client Hardcoded Domain (Domínio codificado do cliente) -->

Como surge Client Hardcoded Domain (Domínio codificado do cliente)?

Os arquivos Javascript podem ser importados dinamicamente de hosts remotos quando são incorporados em HTML. No entanto, essa dependência de um host remoto para esses scripts pode diminuir a segurança, já que os usuários do aplicativo Web são tão seguros quanto o host remoto que serve esses arquivos Javascript.

Exemplo da vulnerabilidade Client Hardcoded Domain (Domínio codificado do cliente):

Na imagem a seguir, o arquivo JavaScript importado em “https://fontsgoogleapiscom” no trecho em vermelho, é de um domínio remoto, o que pode permitir que invasores substituam seu conteúdo por código malicioso.

```

<!DOCTYPE html>
<html lang="pt-br">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>WebGoat - Lista de Vulnerabilidades</title>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link
    href="https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,100;0,200;0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,300;1,400;1,500;1,600;1,700;1,800;1,900&display=swap"
    rel="stylesheet">
  <link rel="stylesheet" href="../../gustavo_projeto_webgoat_web_aplication/css/vulnerabilidades.css">
</head>

<body>
  <div class="buscar">
    <input id="inputBusca" type="text" placeholder="O que você procura?">

    <ul id="listaVulnerabilidades">
    </ul>
    <button class="botao_filtrar_vulnerabilidades" onclick="filtrar()">Filtrar</button>

  </div>

  <h1 class="text_owasp_top_10_vulnerabilidades">OWASP Top 10 Vulnerabilidades - 2021
  <button class="button_ir_para_criar_dados_vulnerabilidades" type="button" onclick="voltarParaCriar()"> Criar Vulnerabilidades</button>
</h1>

  <div id="cards"></div>
  <script src="../../gustavo_projeto_webgoat_web_aplication/js/filtrar.js"></script>

  <script>
    function voltarParaCriar() {
      window.location.href = 'criar_dados_vulnerabilidades.html';
    }
  </script>
</body>
</html>

```

Recomendações gerais:

Como evitá-lo?

- Sempre que possível, hospede todos os arquivos de script localmente, em vez de remotamente. Certifique-se de que os arquivos de script 3rd party hospedados localmente sejam constantemente atualizados e mantidos.

Vulnerabilidade Client Cookies Inspection (Inspeção de Cookies do Cliente) -->

Como surge Client Cookies Inspection (Inspeção de Cookies do Cliente)?

Informações confidenciais estão sendo armazenadas em um cookie por código de cliente.

Exemplo da vulnerabilidade Client Cookies Inspection (Inspeção de Cookies do Cliente):

Na imagem a seguir, os dados confidenciais na senha do trecho em amarelo, são armazenados no cookie no trecho em vermelho.

```

function login() {
  const login = document.getElementById('loginLogin').value;
  const senha = document.getElementById('loginSenha').value;
  const url = `http://localhost:3000/login/${login}/${senha}`;

  fetch(url)
    .then(response => response.json())
    .then(data => {
      if (data.success) {
        console.log("User logged in:", data.userInfo);

        document.cookie = `sessionID=${data.sessionID}; path=/`;
        console.log("Session ID:", data.sessionID);

        window.location.href = 'filtrar_vulnerabilidades.html';
      } else {
        console.error("Login failed:", data.message);
      }
    })
    .catch(error => console.error("Error:", error));
}

```

Recomendações gerais:

Como evitá-lo?

- Nunca armazene informações confidenciais em cookies, ou em qualquer outro lugar do cliente.

