RELATÓRIO 6 – CLIENT SIDE EXPLOITATION – PARTE 2

Lucas Loscheider Reis Muniz – lucaslc01@hotmail.com

Clickjacking

Segundo o site OWASP, é quando um atacante usa várias camadas opacas ou transparentes para enganar um usuário a clickar em um botão ou link em outra página quand, na verdade o objetivo era clickar na página visível, não transparente, não opaca. Assim o atacante “sequestra” o click do usuário. Por exemplo, um usuário que pretende clickar no link escrito “click aqui para ganhar um celular”, estará clickando em um link de uma página por cima da anterior, a qual esta página por cima está transparente, escrito “deletar todas as mensagens”, de uma página de email invisível, sendo esse email um email qualquer de alguém, por exemplo.

Para se defender contra clickjacking podemos:

- Empregar um código defensivo na interface do usuário (UI) para garantir que o frame (ou página) atual é o de maior destaque dentre as páginas visíveis ou invisíveis na mesma janela do navegador;
- Mandar a apropriada Política de Segurança de Conteúdo (CSP) do envio dos cabeçalhos de resposta que instruem o navegador a não permitir frames ou páginas de outros domínios. O CSP permite que você especifique quais URLs relacionados podem enquadrar o recurso atual. Usando a diretiva CSP dos frame-ancestores podemos bloquear ou permitir que uma página seja colocada dentro de um frame ou iframe.

Vulnerabilidades DOM-based

Segundo o site developer.monzilla.org, Document Object Model (DOM) é uma interface para os documentos HTML e XML a qual representa uma página web que pode ter sua estrutura, estilo e conteúdo alterados, por outros programas. Dessa forma o DOM é uma representação orientada a objetos da página web, que pode ser modificada com uma linguagem de script como JavaScript. O problema é que códigos em JavaScript, por exemplo, que estejam incompletos ou errados, pode manipular informações de forma insegura, permitindo assim ataques para controlar valores e funções.

De acordo com o site portswigger.net, não existe uma única ação para eliminar completamente ataques DOM-based. Porém uma forma efetiva para evitar esses ataques é tentar evitar permitir que datas e informações de qualquer fonte não confiável possam alterar dinamicamente o valor que é transmitido para funções JavaScript ou objeto DOM. Também podemos criar listas já verificadas de fontes confiáveis as quais tem permissão de mandar dados e informações que alterem dinamicamente funções JavaScript e objetos DOM.

Cross-origin resource sharing (CORS)

De acordo com o site portswigger.net, CORS é um mecanismo do browser que permite acessos controláveis localizados fora de um dado domínio. Existem as same-origin policy (SOP) que são mecanismos de proteção de um web browser que objetivam restringir scripts de um site origem a acessarem informações de outros sites, prevenindo assim websites atacarem outros. Porém, caso o SOP de um site esteja mal configurado e implementado, ataques CORS podem acontecer.

Exemplo: Normalmente um cliente web manda um pedido para obter um recurso de um domínio diferente do domínio atual. Esse cliente informa, ao servidor, a fonte do outro domínio usando um HTTP header request “Origin”. Depois disso, a aplicação web responde ao pedido, informando ao cliente web os domínios permitidos, usando um HTTP response header Access-Control-Allow-Origin. Posteriormente, o cliente web processa a resposta da aplicação web. Nesse momento, é decisão do cliente web (geralmente o navegador) determinar, usando a resposta da aplicação web HTTP header Access-Control-Allow-Origin, se o cliente web pode acessar os dados do outro domínio.

Em vista disso, o risco do ataque em CORS é que outro cliente web do navegador (além do cliente original) pode (usando, por exemplo, um Burp Suite) para mudar o valor no pedido HTTP de origem (Origin) com o objetivo de forçar a aplicação web a prover o conteúdo do domínio alvo.

Para evitar esse tipo de ataque podemos:

- Caso uma fonte web contenha informação sigilosa, a fonte origem deve ser especificada no Access-Control-Allow-Oring, header;
- Domínios de origem confiável devem ser especificados no Access-Control-Allow-Origin header;
- Evitar usar wildcards (certificado SSL, Security Sockets Layer, para a permissão de acesso de vários subdomínios de um mesmo domínio origem) na rede interna.

XML external entity (XXE) injection

É uma vulnerabilidade, segundo o site portswigger, que permite que um atacante interfira no processamento de uma aplicação XML. XML (Extensible Markup Language) é uma linguagem de marcação usada para definir padrões e formatos de exibição dentro de um documento.
Em vista disso essa vulnerabilidade permite que o atacante vizualize arquivos no filesystem do servidor de aplicação e interaja com qualquer back-end ou sistema externo que a própria aplicação pode acessar. Algumas aplicações usam o formato XML para transmitir informações entre o browser e o servidor.

Como consequências teremos vizualização de informações confidenciais, DoS (negação de serviço), SSRF (explicado posteriormente) e outros.

Para previnir é importante desativar todas as funcionalidades padrões das applications’s XML parsing library que não serão usadas na sua aplicação. Consulte a documentação do seu XML parsing library para desativar funções desnecessárias.

SSRF - Server Side Request Forgery

É um ataque o qual, de acordo com o site portswigger.net, o atacante induz uma aplicação do servidor a fazer requisições HTTP para um domínio arbitrário à escolha do atacante. Com isso, ações não autorizadas ou acessos a informações da organização podem ser feitos.

Por exemplo, supondo que uma aplicação de compras permite um usuário ver se um determinado produto está em um estoque. Para tal informação, a aplicação deve consultar vários APIs de back-end, dependendo do produto e do estoque. Com esse objetivo, a função é implementada enviando o link URL para o API back-end relevante, usando uma requisição front-end HTTP, para assim obter o status do stock e retornar ao usuário. Nessa situação o atacante pode modificar o link URL enviado para um local, como por exemplo algum local do próprio admin do servidor para obter acesso. Com isso, o atacante obtém esse acesso, pois a requisição URL está vindo de, teoricamente, de uma máquina local, ou seja, de um usuário autorizado.

Para evitar, precisamos saber qual funcionalidade da aplicação e requerimentos estão sendo utilizados. De acordo com o site cheatssheetseries.owasp.org, existem 2 casos básicos que um SSRF pode acontecer:

- Quando apenas aplicações confiáveis e conhecidas precisam ser controladas pelo servidor: Nesse caso, as chamadas whitelists (listas com endereços confiáveis) precisam ser usadas. Também deve ser garantido, a nível de conexão, que apenas os servers necessários sejam acessíveis.

- Quando domínios externos e endereços IP não possam ser restringidos: Nesse caso uma blacklist (lista de domínios perigosos e vulneráveis) pode ser usada. Porém ela não é tão eficiente.

Laboratórios resolvidos

De todos os exercícios, apenas o de **Stored DOM XSS** eu não gravei o vídeo porque mesmo pesquisando e vendo algumas soluções, não consegui entender o seu funcionamento/processo de execução. Então não faria sentido gravar um vídeo contendo a resposta sendo que não saberia explicar como ocorre.
Mais uma vez, me desculpe pelo áudio do cooler do notebook nos vídeos. Eu não consegui diminuir esse ruído.

Basic SSRF against the local server: https://youtu.be/Tvg8V67ZOCw
Clickjacking with form input data prefilled from a URL parameter: https://youtu.be/4v0TnMlsBCI
CORS vulnerability with basic origin reflection: https://youtu.be/ekAlkPvBqI4
DOM based open redirection: https://youtu.be/5kUtIXv4PPc
Exploiting XXE to perform SSRF attacks: https://youtu.be/CLQ1hjwPdiA