OWASP Top 10

Java EE

Os 10 riscos mais críticos para aplicações web baseadas em Java EE

Sumário

Introdução

Bem vindos ao OWASP Top 10 2010 para Java EE! Esta é uma edição traduzida e atualizada do OWASP Top 10 2007 for Java EE, que enumera os riscos mais sérios existentes em aplicações web, mostra como se proteger deles e provê links para maiores informações. Este documento utiliza o OWASP Top 10 2010 como base, mas o conteúdo é reescrito e ajustado para falar apenas de aplicações Java EE.

Objetivo

O objetivo principal do OWASP Top 10 para Java EE é educar desenvolvedores Java, designers, arquitetos e organizações sobre as conseqüências das vulnerabilidades de segurança mais comuns em aplicações Java EE. O Top 10 provê métodos básicos de proteção contra estas vulnerabilidades – um ótimo começo para o seu programa de desenvolvimento seguro.

Segurança não ocorre apenas uma vez. É insuficiente proteger seu código apenas uma vez. Em 2012, este Top 10 para Java EE terá mudado e se não mudar uma linha no código da sua aplicação, você poderá estar vulnerável. Por favor, veja os conselhos em Pra onde ir a partir daqui para maiores informações.

Uma iniciativa de desenvolvimento seguro deve lidar com todos os estágios do ciclo de vida de um programa. Aplicações Java EE são possíveis apenas quando um SDLC (Ciclo de Vida de Desenvolvimento Seguro em português) é utilizado. Programas seguros são seguros por design, durante o desenvolvimento e por padrão.

Existem pelo menos 300 problemas que afetam a segurança em uma aplicação web. Estes problemas são detalhados no OWASP Guide que é uma leitura essencial para qualquer um que desenvolva aplicações web hoje em dia.

Este documento é primeiramente educacional, não um padrão. Por favor, não adote este documento como uma política ou um padrão sem falar conosco primeiro! Se você precisa de uma política ou um padrão de programação segura, a OWASP possui projetos sobre estes temas em andamento. Por favor, considere participar ou ajudar financeiramente estes projetos.

Outro projeto interessante da OWASP é o projeto OWASP Code Review onde você irá aprender como auditar suas aplicações Java EE por vulnerabilidades de segurança examinado o código fonte.

Agradecimentos

Nós agradecemos o MITRE por criar a Distribuição de Tipos de Vulnerabilidades no CVE disponível para todos gratuitamente. O projeto OWASP Top 10 é conduzido e patrocinado por Aspect Security.

Líder de Projeto: Dave Wichers

Contribuidores: Jeff Williams

Andrew van der Stock

O OWASP Top 10 para Java EE foi criado por Erwin Geirnaert (ZION Security, OWASP Belux Board) e patrocinado pelo projeto OWASP Spring of Code. E esta versão em português foi traduzida e atualizada por Magno Rodrigues (OWASP Paraiba Chapter Leader).

Gostaríamos de agradecer aos nossos revisores:

Versão original:

Shreeraj Shah (Blueinfy Solutions)
Andrea Cogliati
Jeff Williams (Aspect Security, OWASP Foundation)
Yiannis Pavlosoglou (Information Risk Management PLC)
John Wilander (OmegaPoint)

Versão em português:

Manuela Andrade

Resumo

A1 – Injeção	Falhas de injeção, particularmente SQL Injection são comuns em aplicações Java EE. Injeções ocorrem quando dados informados pelo usuário são enviados para um interpretador como parte de um comando ou uma query. O dado malicioso do atacante faz o interpretador executar comandos que não deveria ou modificar dados.
A2 – Cross Site Scripting (XSS)	Falhas de XSS ocorrem sempre quando uma aplicação Java EE recebe dados informados pelo usuário e envia para o navegador sem validá-los ou codificá-los antes. XSS permite que atacantes executem script no browser da vítima que podem seqüestrar sessões de usuários, desfigurar (deface) web sites e possivelmente instalar malwares, etc.
A3 - Autenticação Falha e Gerenciamento de Sessão	Credenciais de contas e tokens de sessões não são protegidos de forma correta. Atacantes podem comprometer senhas, chaves ou tokens de autenticação para assumir identidades de outros usuários.
A4 – Referência Insegura e Direta a Objetos	Uma referência direta à objeto ocorre quando o desenvolvedor expõe uma referência à uma implementação interna do objeto, como um arquivo, diretório, registro do banco de dados ou chave como uma URL ou parâmetro de formulário. Atacantes podem manipular essas referências para acessar objetos sem autorização.
A5 - Cross Site Request Forgery (CSRF)	Um ataque CSRF força o navegador de uma vitima logada a enviar um pedido pré-autenticado para uma aplicação Java EE vulnerável, que então força ao navegador da vítima a realizar uma ação hostil para o benefício do atacante. CSRF pode ser tão poderoso quando a aplicação web que ele ataca.
A6 – Falhas de Configuração de Segurança	Aplicações podem vazar informações de forma não intencional sobre suas configurações, funcionamento interno ou violar a privacidade através de uma gama variada de problemas. Atacantes usam esta falha para roubar dados sensíveis e conduzir ataques mais sérios.
A7 – Armazenamento Criptográfico Inseguro	Aplicações Java EE raramente utilizam de funções criptográficas de forma correta para proteger dados e credenciais. Atacantes usam dados fracamente protegidos para conduzir roubo de identidade e outros crimes, como fraudes bancárias.
A8 – Falha na Restrição de Acesso à URLs	Frequentemente, uma aplicação Java EE protege apenas funcionalidade sensíveis prevenindo a exibição de links ou URLs para usuários não autorizados. Atacantes podem usar esta falha para acessar e realizar operações não autorizadas pelo acesso dessas URLs diretamente.
A9 – Proteção Insuficiente na Camada de Transporte	Aplicações Java EE frequentemente falham ao criptografar o trafego da rede quando é necessário para proteger comunicações sensíveis.

A10 – Redirects e Forwards não validados	Aplicações web podem redirecionar ou enviar usuários para outras páginas ou web sites e utilizar dados não-confiáveis para determinar a página de destino. Sem a validação adequada, atacantes podem redirecionar as vítimas para sites de phishing ou que contenham malware ou utilizar os forwards para acessar páginas não autorizadas.

Tabela 1: Top 10 de vulnerabilidades em aplicações web de 2010

A1 – FALHAS DE INJEÇÃO

Falhas de injeção, particularmente SQL Injection, são comuns em aplicações Java EE. Existem diversos tipos de injeção: SQL, LDAP, XPath, XSLT, HTML, XML e muito mais.

A Injeção ocorre quando dados informados pelo usuário são enviados para o interpretador como parte de um comando ou uma query em particular. Atacantes enganam o interpretador em executar comandos não intencionados através do fornecimento de dados especialmente fabricados. Falhas de injeção permitem que atacantes criem, leiam, atualizem ou apaguem qualquer dado disponível para a aplicação. No pior dos cenários, estas falhas permitem que um atacante comprometa completamente a aplicação e os sistemas sob ela, contornando até mesmo ambientes protegidos com diversos firewalls.

AMBIENTES AFETADOS

Todos os frameworks de aplicações Java EE que utilizam interpretadores ou chamam outros processos estão vulneráveis a ataques de injeção. Isto inclui qualquer componente do framework que possa utilizar interpretadores de back-end.

VULNERABILIDADES

Se os dados informados pelo usuário são passados para um interpretador sem uma validação ou codificação apropriada, a aplicação está vulnerável. Verifique se dados de entrada são fornecidos para queries dinâmicas, como:

Java:

```
String query = "SELECT user_id FROM user_data WHERE user_name = ' " +
req.getParameter("userID") + " ' and user_password = ' " +
req.getParameter("pwd") +" ' ";
```

No exemplo, o framework não validou os dados informados pelo usuário então era possível realizar o login com *') or '1'='1'* -- para o login e a senha, um exemplo clássico de SQL Injection.

```
Runtime.exec( "C:\\windows\system32\cmd.exe \C netstat -p " +
req.getParameter("proto");
```

Este exemplo mostra o Sistema Operacional executando uma shell de commando com parâmetros não validados, permitindo um atacante informar "udp; format C:" para apagar o disco rígido da aplicação. Todos os interpretados estão sujeitos à injeção se a aplicação inclui dados informados pelos usuários no comando.

VERIFICANDO A SEGURANÇA

O objetivo é verificar que os dados do usuário não podem modificar o significado dos comandos e das queries enviadas para qualquer dos interpretadores utilizados pela aplicação.

Abordagem automatizada: Diversas ferramentas de análise de vulnerabilidades procuram por problemas de injeção, especialmente SQL Injection. Ferramentas de análise estática que pesquisam por utilização insegura da API de interpretadores são importantes, mas normalmente não conseguem verificar que uma validação ou codificação apropriada está sendo utilizada para proteger contra a vulnerabilidade. Se a aplicação captura erros internos do servidor ou erros detalhados do banco de dados, ela pode impedir significativamente ferramentas automatizadas, mas o código pode ainda estar vulnerável. Ferramentas automatizadas podem ser capazes de detectar injeções de LDAP, XML e XPath.

Abordagem manual: A abordagem mais eficiente e precisa é auditar o código que chama os interpretadores. O revisor ou auditor deve verificar o uso de uma API segura ou que uma validação e/ou codificação está sendo utilizada. Testes podem ser extremamente demorados porque a superfície de ataque da maioria das aplicações é muito grande.

PROTEÇÃO

Evite o uso de interpretadores sempre que possível. Se você necessita utilizar um interpretador, a maneira chave para evitar injeções é utilizar APIs seguras como queries fortemente parametrizadas e bibliotecas de mapeamento objeto-relacional como HIbernate (hibernate.org). Estas interfaces fazem todo o escape de dados ou não precisam de escape. Note que mesmo com as interfaces resolvendo o problema, a validação ainda é recomendada para detectar ataques.

Utilizar interpretadores é perigoso, então vale a pena tomar um cuidado maior, como os seguintes:

- **Validação de entrada.** Utilize um mecanismo de validação de entrada padrão para validar todos os dados em tamanho, tipo, sintaxe e regras de negócio antes de exibi-los ou armazená-los. Use a estratégia de validação do tipo whitelist. Rejeite entradas inválidas ao invés de tentar sanitizar dados maliciosos. Não se esqueça que mensagens de erro também podem incluir dados inválidos.
- Utilize APIs fortemente parametrizadas para as queries, mesmo quando for chamar stored procedures.
- Imponha o menor privilégio quando se conectar a banco de dados ou outros sistemas de back-end
- Evite mensagens de erro detalhadas que podem ser úteis para um atacante
- Use stored procedures já que elas geralmente são protegidas de SQL Injection.
- Não utilize interfaces de query dinâmicas (como executeQuery() ou similares)

- Não use funções de escape simples, ao invés disso, use PreparedStatement.
- Quando utilizar mecanismos simples de escape, lembre-se que estas funções não conseguem "escapar" nomes de tabelas! Nomes de tabelas devem ser SQL permitidos e por isso são completamente impróprias para dados informados pelo usuário
- **Cuidado com erros de canonicalização**. Entradas devem ser decodificadas e canonicalizadas para a representação interna da aplicação antes de serem validadas. Certifique-se de que sua aplicação não decodifique a mesma entrada duas vezes. Erros como esse podem ser utilizados para contornar esquemas de whitelist introduzindo entradas perigosas depois que elas foram checadas.

Recomendações específicas da linguagem:

- Java EE – utilize PreparedStatements fortemente tipados ou mapeamento objeto relacional como Hibernate ou Spring.
- Utilize as classes Encoder e Validator da OWASP ESAPI (Enterprise Security API)

```
String input = request.getParameter("param");
if (input != null) {
   if (!Validator.getInstance().isValidString("^[a-zA-Z ]*$", input)) {
        response.getWriter().write("Inválido: " +
        Encoder.getInstance().encodeForHTML(input) + "<br>");
   } else {
        response.getWriter().write("Válido: " +
        Encoder.getInstance().encodeForHTML(input) + "<br>");
   }
```

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5121
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4953
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4592

REFERÊNCIAS

- CWE: CWE-89 (SQL Injection), CWE-77 (Command Injection), CWE-90 (LDAP Injection), CWE-91 (XML Injection), CWE-93 (CRLF Injection), others.
- WASC Threat Classification:
  http://www.webappsec.org/projects/threat/classes/ldap_injection.shtml
  http://www.webappsec.org/projects/threat/classes/sql_injection.shtml
  http://www.webappsec.org/projects/threat/classes/os_commanding.shtml
- OWASP, http://www.owasp.org/index.php/SQL_Injection
- OWASP Guide, http://www.owasp.org/index.php/Guide_to_SQL_Injection
- OWASP Code Review Guide,
  http://www.owasp.org/index.php/Reviewing_Code_for_SQL_Injection
- OWASP Testing Guide,
  http://www.owasp.org/index.php/Testing_for_SQL_Injection

- SQL Injection, http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
- Advanced SQL Injection, http://www.ngssoftware.com/papers/advanced_sql_injection.pdf
- More Advanced SQL Injection, http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf
- Hibernate, an advanced object relational manager (ORM) for Java EE and .NET, http://www.hibernate.org/
- Java EE Prepared Statements, http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A2 – CROSS SITE SCRIPTING (XSS)

Cross Site Scripting, mais conhecido como XSS, é de fato um subgrupo de HTML Injection. XSS é o problema de segurança em aplicações web mais prevalente e nocivo. Falhas de XSS ocorrem quando uma aplicação Java EE recebe dados originados de um usuário e os envia para um navegador web sem validar ou codificar previamente aquele conteúdo.

XSS permite que atacantes executem scripts no browser das vítimas que podem sequestar sessões de usuários, desfigurar web sites, inserir conteúdo hostil, conduzir ataques de phishing e controlar o navegador do usuário usando malware. O script malicioso normalmente é JavaScript, mas qualquer linguagem de script suportada pelo browser da vítima é um alvo em potencial para este ataque.

AMBIENTES AFETADOS

Todos os frameworks de aplicações Java EE são vulneráveis ao Cross Site Scripting. Struts teve problemas de XSS nas páginas de erro embutidas e servidores de aplicação têm problemas com páginas de erros, consoles administrativos e exemplos.

VULNERABILIDADES

Existem três tipos de XSS conhecidos: refletido (ou não persistente), stored (ou persistente) e baseado em DOM. XSS Refletido é o mais fácil de ser explorado – uma página irá refletir dados informados pelo usuário de volta para o usuário mostrado no seguinte pedaço de código. A págima HTML irá retornar as palavras digitadas na pesquisa, não validadas, para o usuário:

out.writeln(“Você pesquisou por: “+request.getParameter(“query”);

De outra forma em uma página JSP:

<%=request.getParameter(“query”);%>

O XSS Stored obtém os dados maliciosos, armazena-os em um arquivo, banco de dados ou outro sistema de back end e depois, em um outro momento, exibe os dados para o usuário, não validados. Isto é extremamente perigoso em sistemas como CMS, blogs ou fóruns onde um grande número de usuários irão visualizar informações colocadas por outros indivíduos. Neste trecho de código, dados são recuperados do banco de dados e retornados na página HTML sem nenhuma validação:

```
out.writeln("<tr><td>" + visitante.nome + "<td>" + visitante.comentario);
```

Com ataques XSS baseados em DOM, o código JavaScript e as variáveis do site são manipuladas ao invés de elementos HTML. Um exemplo fácil de um HTML vulnerável em uma aplicação pode ser encontrando no artigo referenciado abaixo de Amit Klein:

```
<HTML>
<TITLE>Bem vindo!</TITLE>Hi<SCRIPT>
var pos=document.URL.indexOf("nome=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Bem vindo ao nosso sistema…
</HTML>
```

Alternativamente, ataques podem ser uma mistura dos três tipos. O perigo do XSS não é que tipo de ataque é explorável, mas se é possível injetar um payload malicioso. Comportamentos fora do padrão ou inesperado de navegadores podem introduzir vetores de ataques sutis. XSS também é alcançado através de qualquer componente que o seu navegador utilize, por exemplo, Watchfire descobriu uma vulnerabilidade XSS no Google Desktop, que é um componente integrado no seu browser.

Ataques são normalmente implementados em JavaScript, que é uma poderosa linguagem de script. Utilizar o JavaScript pode permitir aos atacantes manipular qualquer aspecto da página renderizada. Estas manipulações podem ser: adicionar novos elementos (como um campo de login e senha que envia as credenciais para um site hostil), manipular qualquer aspecto da árvore DOM interna ou modificar a maneira que a página se comporta. JavaScript permite o uso de XmlHttpRequest que é típicamente utilizado por sites que implementam a tecnologia AJAX (http://java.sun.com/developer/technicalArticles/J2EE/AJAX/).

Utilizando o XmlHttpRequest é possível contornar a Política de Mesma Origem (SOP em inglês) de um navegador para encaminhar a vítima para um site malicioso. Fortify descobriu uma vulnerabilidade específica com JavaScript e chamou de JavaScript Hijacking. Esta técnica pode permitir a criação de worms complexos e zumbis maliciosos que duram enquanto o navegador estiver aberto. Ataques ao AJAX não precisam ser visíveis e não necessitam a interação com usuário para realizar ataques Cross Site Request Forgery (CSRF) perigosos (veja A5).

Mais informações sobre Cross Site Scripting e detalhes técnicos sobre exploração de XSS podem ser encontrados no livro XSS Exploits.

VERIFICANDO A SEGURANÇA

O objetivo é verificar se todos os parâmetros na aplicação são validados e/ou codificados antes de serem incluídos nas páginas HTML.

Abordagem automatizada: Ferramentas de testes de invasão automatizadas são capazes de detectar XSS refletido através de injeção de parâmetros, mas normalmente falham ao encontrar XSS persistentes, particularmente se a saída do vetor XSS injetado for prevenida via checagens de autorização. Ferramentas automatizadas de auditoria de código podem encontrar chamadas fracas ou perigosas à API, mas normalmente não podem determinar o nível de validação ou codificação que foi utilizado. Isto

resulta num grande número de falsos positivos. Ferramentas mais modernas e comerciais de análise estática são capazes de realizar análises de fluxo de dados inter-processos e são configuráveis para que elas possam reconhecer métodos de validação e reduzir drasticamente a quantidade de falsos positivos. Atualmente, existe apenas uma única ferramenta disponível publicamente para análise e identificação de XSS baseado em DOM que é o DOMinator, que funciona como um plugin para o Firefox e foi desenvolvido pelo Stefano Di Paola, um membro da OWASP.

Abordagem manual: Se um mecanismo centralizado de validação e codificação é utilizado, a maneira mais eficiente para verificar a segurança é auditar o código. Se uma implementação distribuída é utilizada, então a verificação poderá ser consideravelmente mais demorada. Testes também são demorados pois a superfície de ataque na maioria das aplicações é muito grande.

PROTEÇÃO

A melhor proteção para XSS é uma combinação de validação whitelist de todos os dados de entrada e codificação apropriada de todos os dados de saída. Validação permite a detecção de ataques e a codificação previne que qualquer injeção de script rode no navegador com sucesso. Evitar o XSS em toda uma aplicação requer uma abordagem arquitetural consistente:

- **Validação de entrada.** Utilize um mecanismo padrão de validação de entrada para validar todos os dados de entrada em tamanho, tipo, sintaxe e regras de negócio antes de permitir que os dados sejam exibidos ou armazenados. Use a estratégia de validação whitelist (aceitar apenas os permitidos). Rejeite entradas inválidas ao invés de tentar sanitizar dados maliciosos. Não se esqueça que mensagens de erro também podem incluir dados inválidos.
- **Codificação de saída.** Certifique-se que todos os dados informados pelo usuário são propriamente codificados antes de renderizá-los, utilizando a abordagem de codificar todos os caracteres ao invés de apenas um subgrupo muito limitado.
- **Especifique a codificação de saída** (como ISSO 8859-1 ou UTF 8). Não permita que o atacante escolha isso para os seus usuários.
- **Não utilize validação do tipo blacklist** para detectar XSS na entrada ou codificar a saída. Pesquisar e substituir apenas alguns caracteres (“<”, “>” e outros similares ou palavras como “script”) é um método fraco e já foi contornado com sucesso diversas vezes. Até mesmo a tag “<b>” não é segura em alguns contextos. XSS tem um número surpreendente de variantes o que torna fácil contornar a validação do tipo blacklist.
- **Cuidado com erros de canonicalização**. Entradas devem ser decodificadas e canonicalizadas para a representação interna da aplicação antes de serem validadas. Certifique-se de que sua aplicação não decodifique a mesma entrada duas vezes. Erros como esse podem ser utilizar para contornar esquemas de whitelist introduzindo entradas perigosas depois que elas foram checadas.

Recomendações específicas para o Java EE:

- **Validação de entrada, lado servidor:**
  Utilize os validadores do Struts para validar todas as entradas
  Implemente expressões regulares em Java para validar as entradas
  Use a validação do JSF no lado servidor:
  f:validadeLength para o tamanho permitido da entrada:
  Ex.: <f:validateLength minimum="2" maximum="10"/>
  <h:inputText required=”true”> se um campo de entrada for obrigatório
- **Codificação da saída:**
  Utilize os mecanismos de saída do Struts como o <bean:write ...> ou use o atributo padrão da JSTL escapeXML=”true” na <c:out ...>. **Não** utilize <%= ...%> fora de um mecanismo apropriado de codificação.
- **Utilize as classes Encoder e Validator da OWASP ESAPI** (Enterprise Security API)

```
if ( !Validator.getInstance().isValidHTTPRequest(request) ) {
        response.getWriter().write( "<P>HTTP Request Inválido – Caracteres
    Inválidos</P>" );
}
```

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4206
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3966
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5204

REFERÊNCIAS

- CWE: CWE-79, Cross-Site scripting (XSS)
- WASC Threat Classification:
  http://www.webappsec.org/projects/threat/classes/crosssite_scripting.shtml
- OWASP – Cross site scripting,
  http://www.owasp.org/index.php/Cross_Site_Scripting
- OWASP – Testing for XSS,
  http://www.owasp.org/index.php/Testing_for_Cross_site_scripting
- OWASP Stinger Project (A Java EE validation filter) –
  http://www.owasp.org/index.php/Category:OWASP_Stinger_Project
- OWASP PHP Filter Project -
  http://www.owasp.org/index.php/OWASP_PHP_Filters
- OWASP Encoding Project -
  http://www.owasp.org/index.php/Category:OWASP_Encoding_Project
- RSnake, XSS Cheat Sheet, http://ha.ckers.org/xss.html
- Klein, A., DOM Based Cross Site Scripting,
  http://www.webappsec.org/projects/articles/071105.shtml

- .NET Anti-XSS Library - http://www.microsoft.com/downloads/details.aspx?FamilyID=efb9c819-53ff-4f82-bfaf-e11625130c25&DisplayLang=en
- XSS Exploits - http://www.amazon.com/Cross-Site-Scripting-Attacks-Exploits/dp/1597491543
- Watchfire Google Desktop XSS - http://download.watchfire.com/whitepapers/Overtaking-Google-Desktop.pdf
- Fortify Javascript Hijacking - http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A3 – FALHAS DE AUTENTICAÇÃO E GERÊNCIA DE SESSÕES

Um gerenciamento de sessões e de autenticação apropriados são críticos para a segurança de aplicações web. Falhas nesta área frequentemente envolvem falha ao proteger credenciais e tokens de sessão durante seu ciclo de vida. Estas falhas podem levar ao seqüestro de contas do usuário ou administrador, manipulação dos controles de autorização e causar violação de privacidade.

AMBIENTES AFETADOS

Todos os frameworks de aplicações Java EE são vulneráveis as falhas de autenticação e de gerência de sessões.

VULNERABILIDADES

Falhas no principal mecanismo de autenticação não são raras, mas vulnerabilidades são introduzidas mais frequentemente através de funções de autenticação auxiliares como logout, administração de senha, timeout, “lembrar-me”, pergunta secreta e atualização da conta.

VERIFICANDO A SEGURANÇA

O objetivo é verificar se a aplicação autentica de forma apropriada os usuários e protege as identidades e suas credenciais.

Abordagem automatizada: Ferramentas de análise de vulnerabilidades possuem muita dificuldade em detectar falhas em mecanismos de autenticação e gerência de sessões customizados. Ferramentas de análise estática provavelmente não detectarão problemas desse tipo no código.

Abordagem manual: Revisão de código e testes, especialmente combinados, são bastante efetivos em verificar se a autenticação, a gerência das sessões e as funções auxiliares estão todas implementas de forma correta.

PROTEÇÃO

Autenticação depende da comunicação e do armazenamento seguro das credenciais. Primeiramente garanta que SSL é a única opção para todas as partes autenticadas da aplicação (veja A9 – Proteção Insuficiente na Camada de Transporte) e todas as credenciais são armazenadas na forma de hash ou criptografadas (veja A8 – Armazenamento Criptográfico Inseguro).
Prevenir falhas de autenticação requer um planejamento cauteloso. Algumas das mais importantes considerações são:

- Uma das coisas mais importante de se implementar é um sistema de logs auditável para controles de autenticação e autorização. Você deve ser capaz de responder as seguintes questões facilmente:

    - Quem logou?
    - Quando?
    - De onde?
    - Que transações o usuário iniciou?
    - Quais dados foram acessados?
- Utilize apenas o mecanismo de gerência de sessões embutido. Não escreve ou use controladores de sessão secundários sob nenhuma circunstância
- Não aceite identificadores de sessão novos, pré-definidos ou inválidos da URL ou no request. Isto é chamado de Session Fixation Attack.
- Use um único mecanismo de autenticação suficientemente forte e com vários fatores. Certifique-se de que este mecanismo não está facilmente sujeito a ataques de Spoofing ou Replay. Não o faça complexo demais o que pode torná-lo vulnerável ao seu próprio ataque.
- Implemente uma forte política de senhas para os usuários. Isto irá prevenir que senhas fáceis de serem descobertas como palavras do dicionário.
- Não permita que o processo de login inicie de uma página não criptografada. Sempre inicie o processo através de uma página com um novo cookie de sessão para prevenir roubo de sessões ou credenciais, ataques de phishing e session fixation attacks.
- Considere gerar uma nova sessão depois de uma autenticação com sucesso ou mudança de privilégios.
- Verifique se todas as páginas têm um link para o logout. O logout deve destruir todo o estado da sessão no lado servidor e os cookies no lado cliente. Considere fatores humanos: não pergunte por confirmação ou usuários irão acabar fechando a aba ou a janela ao invés de realizar o logout de forma apropriada.
- Use um período de timeout que automaticamente realiza o logout de uma sessão inativa
- Utilize apenas fortes funções auxiliares (perguntas e respostas, reset da senha) já que estas são credenciais da mesma forma que nomes de usuário, senhas e tokens são. Aplique hash nas respostas para evitar ataques de disclosure.
- Não exponha nenhum identificador de sessão ou qualquer porção de credenciais válidas nas URLs ou logs (nada de session rewriting ou armazenar a senha do usuário nos arquivos de log)
- Exija que usuário informe a senha antiga quando ele decidir trocar a senha.
- Não confie em credenciais que podem ser “falsificadas” como única forma de autenticação, como endereços IP ou máscaras de endereço, lookups de DNS ou DNS reverso, cabeçalhos referrer ou similares.
- Tenha cuidado ao enviar segredos para o endereço de email cadastrado (veja RSNAKE01 nas referências) como um mecanismo para o reset de senhas. Utilize números aleatórios temporários para resetar o acesso e envie um email informando que a senha foi alterada. Tenha cuidado ao permitir que usuários mudem seu endereço de email – envie uma mensagem para o email anterior antes de realizar a mudança.
- Adicione uma restrição de segurança no web.xml para todas as URLs que precisarem do HTTPS

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>Páginas em HTTPS</web-resource-name>
                <url-pattern>/profile</url-pattern>
                <url-pattern>/register</url-pattern>
                <url-pattern>/password-login</url-pattern>
                <url-pattern>/ldap-login</url-pattern>
        </web-resource-collection>
        <user-data-constraint>
                <transport-guarantee>CONFIDENCIAL</transport-guarantee>
        </user-data-constraint>
</security-constraint>
```

- Use as classes Authenticator, User e HTTPUtils da OWASP ESAPI

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6145
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6229
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6528

REFERÊNCIAS

- CWE: CWE-287 (Authentication Issues), CWE-522 (Insufficiently Protected Credentials), CWE-311 (Reflection attack in an authentication protocol), others.
- WASC Threat Classification: http://www.webappsec.org/projects/threat/classes/insufficient_authentication.shtml
  http://www.webappsec.org/projects/threat/classes/credential_session_prediction.shtml
  http://www.webappsec.org/projects/threat/classes/session_fixation.shtml
- OWASP Guide, http://www.owasp.org/index.php/Guide_to_Authentication
- OWASP Code Review Guide, http://www.owasp.org/index.php/Reviewing_Code_for_Authentication
- OWASP Testing Guide, http://www.owasp.org/index.php/Testing_for_authentication
- RSNAKE01 - http://ha.ckers.org/blog/20070122/ip-trust-relationships-xss-and-you
- Informit01 – Building a custom JBoss Login Module - http://www.informit.com/articles/article.aspx?p=389111
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A4 – REFERÊNCIA INSEGURA E DIRETA A OBJETOS

Uma referência direta a um objeto ocorre quando um desenvolvedor expõe uma referência para um objeto de implementação interna, como um arquivo, um diretório, registro ou chave do banco de dados, em um parâmetro da URL ou do formulário. Um atacante pode manipular referências diretas a objetos para acessar outros objetos sem autorização, a não ser que um controle de acesso esteja implementado.

Por exemplo, em aplicações de Internet Banking, é comum utilizar o número da conta como chave primária. Portanto, é tentador utilizar o número da conta diretamente na interface web. Mesmo que os desenvolvedores tenham utilizado queries parametrizadas para prevenir SQL Injection, se não existir uma verificação extra de que o usuário é o dono daquela conta e está autorizado a visualizar as informações dela, um atacante pode modificar o parâmetro de número da conta e ver ou alterar todas as contas.

Este tipo de ataque ocorreu com o site da Secretaria da Fazenda da Austrália em 2000, onde um usuário legítimo, mas malicioso simplesmente trocou o ABN (Identificador da Empresa) presente na URL. O usuário obteve do sistema informações de aproximadamente 17 mil empresas, ele então enviou emails para cada uma dessas 17 mil empresas com detalhes do ataque. Este tipo de vulnerabilidade é muito comum, mas não é largamente testado em muitas aplicações.

AMBIENTES AFETADOS

Todos os frameworks de aplicações web são vulneráveis a ataques de referência direta insegura a objetos.

VULNERABILIDADES

Muitas aplicações expõem suas referências internas a objetos para os usuários. Atacantes utilizam a manipulação de parâmetros para alterar as referências e violar a política de controle de acesso. Frequentemente estas referências apontam para sistemas de arquivos e banco de dados, mas qualquer objeto exposto pela aplicação pode estar vulnerável.

Por exemplo, se o código permite que o usuário especifique nomes ou caminhos de arquivos, isto pode permitir que atacantes saiam do diretório da aplicação e acessem outros recursos.

```
<select name="idioma"><option value="fr">Francês</option></select>
…
Public static String idioma = request.getParameter(idioma);
String idioma = request.getParameter(idioma);
RequestDispatcher rd = context.getRequestDispatcher(“main_”+ idioma);
rd.include(request, response);
```

Este código pode ser atacado utilizando uma String como "../../../../etc/passwd%00" utilizando Null Byte Injection (veja o OWASP Guide para mais informações) para acessar qualquer arquivo no sistema de arquivos do servidor web. Similarmente, referências para chaves do banco de dados são frequentemente expostas. Um atacante pode manipular estes parâmetros simplesmente adivinhando ou procurando por outra chave válida. Frequentemente, elas são seqüenciais por natureza. No exemplo abaixo, mesmo que um aplicação não apresente nenhum link para carrinhos não autorizados e que nenhum SQL Injection seja possível, um atacante ainda pode modificar o parâmetro cartID para qualquer carrinho que ele quiser.

```
int cartID = Integer.parseInt( request.getParameter( "cartID" ) );
String query = "SELECT * FROM table WHERE cartID=" + cartID;
```

VERIFICANDO A SEGURANÇA

O objetivo é verificar se a aplicação não permite que referências diretas a objetos sejam manipuladas por um atacante.

Abordagem automatizada: Ferramentas de análise de vulnerabilidades terão dificuldade em identificar quais parâmetros são suscetíveis à manipulação ou se a alteração funcionou. Ferramentas de análise estática não conseguem saber quais parâmetros devem ter verificação de controle de acesso antes de serem utilizados.

Abordagem manual: Revisão de código pode rastrear parâmetros críticos e identificar se eles são suscetíveis à manipulação em muitos casos. Análise de verificação de fronteiras ou fuzzing são boas maneiras de realizar este trabalho. Testes de invasão também podem verificar se a manipulação é possível. Entretanto, ambas as técnicas são demoradas e podem ser inconsistente.

PROTEÇÃO

A melhor proteção é evitar expor referências diretas a objetos para usuários através da utilização de índices, mapa de referência indireta ou outro método indireto que seja fácil de validar. Se uma referência direta a um objeto deva ser utilizada, certifique-se de que o usuário está autorizado antes de utilizá-la.

Estabelecer uma maneira padrão para referenciar os objetos de uma aplicação é importante:

- Evite expor referências de objetos privados para os usuários sempre que possível como: chaves primárias ou nome de arquivos
- Valide qualquer referência de objeto privadas extensivamente com a abordagem whitelist
- Verifique a autorização para todos os objetos referenciados
- Certifique-se de que a entrada não contenha padrões de ataque como ../ ou %00

A melhor solução é utilizar um índice ou um mapa de referências para prevenir ataques de manipulação de parâmetros.

http://www.example.com/application?file=1

Se você necessitar expor referências diretas para estruturas de banco de dados, certifique-se de que suas queries SQL e outros métodos de acesso ao banco de dados permitem a exibição de apenas registros autorizados.

```
try {

        int cartID = Integer.parseInt( request.getParameter( "cartID" ) );

} catch (NumberFormatException e) {

        // Do error handling
}

User user = (User)request.getSession().getAttribute( "user" );
String query = "SELECT * FROM table WHERE cartID=" + cartID + " AND userID="
+ user.getID();
```

Outra solução é verificar a integridade dos parâmetros para verificar se eles não foram alterados. Esta verificação de integridade pode ser adicionada como um parâmetro adicional utilizando técnicas de criptografia ou hashing. Isto é implementado no framework HTTP Data Integrity Validator.

- Use a classe AccessReferenceMap da OWASP Entreprise Security API (ESAPI)

```
// Setup the access reference map for current users in session
HttpSession session = request.getSession();
AccessReferenceMap arm = (AccessReferenceMap)
session.getAttribute("usermap" );

if ( arm == null ) {
        arm = new AccessReferenceMap();
        request.getSession().setAttribute( "usermap", arm );
}

String param = request.getParameter("user");
String accountName = (String)arm.getDirectReference(param);
```

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0329
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4369
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0229

REFERÊNCIAS

- CWE: CWE-22 (Path Traversal), CWE-472 (Web Parameter Tampering)
- WASC Threat Classification: http://www.webappsec.org/projects/threat/classes/abuse_of_functionality.shtml http://www.webappsec.org/projects/threat/classes/insufficient_authorization.shtml
- OWASP Testing Guide, http://www.owasp.org/index.php/Testing_for_business_logic
- OWASP Testing Guide, http://www.owasp.org/index.php/Testing_for_Directory_Traversal
- OWASP, http://www.owasp.org/index.php/Category:Access_Control_Vulnerability
- GST Assist attack details, http://www.abc.net.au/7.30/stories/s146760.htm
- HTTP Data Integrity Validator framework: http://www.hdiv.org
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A5 – CROSS SITE REQUEST FORGERY (CSRF)

Cross Site Request Forgery não é um ataque novo, mas simples e devastador. Um ataque CSRF força o browser da vítima logada a enviar um request para uma aplicação web vulnerável, que então realiza a ação escolhida na conta da vítima.

Esta vulnerabilidade é extremamente difundida, como qualquer aplicação web que:

- Não possua verificações de autorização para ações vulneráveis
- Irá processar uma ação se for possível enviar um login default no request (ex.: http://www.example.com/admin/doSomething.jsp?username=admin&passwd=admin)
- Autoriza requests baseados apenas em credenciais que são automaticamente submetidas como o cookie de sessão se já estiver logado na aplicação ou a funcionalidade "Lembrar de mim" se não estiver logado ou um token Kerberos se parte de uma Intranet participando no logon integrado com o Active Directory estiver em risco.

Infelizmente, hoje em dia, a maioria das aplicações web confia apenas em credenciais enviadas automaticamente como cookies de sessão, credenciais básicas de autenticação, endereços IP de origem, certificados SSL ou credenciais do domínio Windows.

Esta vulnerabilidade também é conhecida por diversos outros nomes incluindo Session Riding, One-Click Attacks, Cross Site Reference Forgery, Hostile Linking e Automation Attack. O acrônimo XSRF também é frequentemente utilizado. A OWASP e o MITRE padronizaram o termo Cross Site Request Forgery e CSRF.

AMBIENTES AFETADOS

Todos os frameworks Java EE de aplicações web estão vulneráveis ao CSRF.

VULNERABILIDADES

Um típico ataque CSRF contra um fórum pode tomar a forma de direcionar o usuário a chamar uma função, como a página de logout da aplicação. A tag a seguir em qualquer página visitada pela vítima irá gerar um request que irá desconectá-la da aplicação:

```
<img src="http://www.example.com/logout.jsp">
```

Se um sistema de um banco permitisse que sua aplicação processasse os requests, como transferência bancária, um ataque parecido poderia permitir:

```
<img src="http://www.example.com/transfer.do?frmAcct=
document.form.frmAcct&toAcct=4345754&toSWIFTid=434343&amt=3434.43">
```

Jeremiah Grossman, na sua palestra na Black Hat em 2006 “Hacking Intranet Sites from the outside”, demonstrou que é possível forçar um usuário a realizar mudanças no seu roteador DSL sem seu consentimento, mesmo que o usuário não saiba que o roteador DSL tenha uma interface web. Jeremiah utilizou a conta padrão para realizar o ataque.

Todos estes ataques funcionam porque a credencial de autorização do usuário (normalmente o cookie de sessão) é automaticamente inclusa nos requests pelo navegador, mesmo que o atacante não a forneça.

Se a tag contendo o ataque puder ser inserida em uma aplicação vulnerável, então a probabilidade de encontrar vítimas logadas é significantemente maior, semelhante ao aumento do risco entre falhas de XSS Stored e Reflected. Falhas de XSS não são necessárias para um ataque CSRF funcionar, embora qualquer aplicação com falhas de XSS está suscetível ao CSRF porque um ataque CSRF pode explorar falhas de XSS para roubar qualquer credencial que não seja enviada automaticamente que pode ter sido utilizada para proteger contra um ataque CSRF. Muitos worms de aplicação vêm utilizando ambas as técnicas em combinação.

Quando estiver construindo defesas contra ataques CSRF, você deve focar também em eliminar as vulnerabilidades de XSS da sua aplicação, pois estas falhas pode ser utilizadas para burlar a maioria das defesas de CSRF que você implementou.

VERIFICANDO A SEGURANÇA

O objetivo é verificar que a aplicação está protegida contra ataques CSRF através da geração e requisição de algum tipo de token de autorização que não é enviado automaticamente pelo browser.

Abordagem automatizada: Poucas ferramentas automatizadas conseguem detectar vulnerabilidades CSRF hoje em dia, mesmo assim a detecção de CSRF é possível para aplicações com mecanismos de varredura suficientemente capazes. Entretanto, se o seu *scanner* de aplicação detectar uma vulnerabilidade XSS e não existir nenhuma proteção contra CSRF, possivelmente você está em risco de ataques CSRF pré-prontos.

Abordagem manual: Testes de invasão são uma maneira rápida de verificar se proteções contra CSRF estão funcionando. Para verificar se o mecanismo é forte e está devidamente implementado, auditar o código é a ação mais eficiente.

PROTEÇÃO

Aplicações devem garantir que elas não confiam em credenciais ou tokens que são enviados automaticamente pelo navegador. A única solução é utilizar um token que o navegador não irá se “lembrar” como um campo oculto (hidden) único ou um parâmetro GET ou POST adicional e único e então incluir automaticamente este token e então incluí-lo automaticamente em todos os pedidos (requests) para a aplicação web. Um ataque CSRF que não utilize este token será evitado.

As seguintes estratégias devem se inerentes em todas as aplicações web:

- Garanta que não existam vulnerabilidades XSS em sua aplicação (veja A1 – Cross Site Scripting)
- Insira tokens personalizados e aleatórios em todos os formulários e URLs que não serão automaticamente enviadas pelo navegador. Por exemplo, o nome e o valor do campo oculto são únicos para cada request.

  ```
  <form action="/transfer.do" method="post">
  <input type="hidden" name="8438927730" value="43847384383">
  …
  </form>
  ```

  E então verifique se o token enviado está correto para o usuário atual. Estes tokens podem ser únicos para aquele método ou página em particular para aquele usuário, ou simplesmente par a toda a sessão. Quanto mais focado for o token para um método e/ou um conjunto de dados maior será a proteção, mas será mais complicado de programar e manter.
- Para dados sensíveis ou transações de valor, re-autentique ou utilize assinatura de transações para garantir que o pedido é genuíno. Implemente mecanismos externos como email ou telefone para verificar pedidos ou notificar o usuário.
- Não utilize requests GET (URLs) para dados sensíveis ou para realizar transações de valor. Use apenas métodos POST quando for processar dados sensíveis do usuário. Entretanto a URL pode conter o token aleatório e com isso criar uma URL única, o que torna o CSRF quase impossível de realizar
- Somente o POST é insuficiente como proteção. Você também deve combiná-lo com tokens aleatórios, autenticação out of band ou re-autenticação para proteger apropriadamente contra CSRF.
- No Struts você pode usar o org.apache.struts2.components.Token que foi inventado para ajudar no problema de submissão dupla.
- O framework HTTP Data Integrity Validator adiciona um parâmetro aleatório em toda URL ou formulário. Se este não for inserido no request, o request é negado pelo framework.
- Verifique o cabeçalho Content-Type para proteger chamada a funções Ajax e Web Services
- Embora o cabeçalho HTTP Referer possa ser alterado, verificá-lo é uma boa prática para detectar tentativas de hacking.
- Use a classe User da OWASP ESAPI para gerar e validar um token CSRF

  ```
  try {
      HTTPUtilities.getInstance().checkCSRFToken( request );
  } catch( IntrusionException e ) {
      response.getWriter().write( "<P>Request HTTP Inválido - Token CSRF
      não inserido</P>" );
  }
  String valid =
        HTTPUtilities.getInstance().addCSRFToken("/ESAPITest/test?param=test");
  response.getWriter().write(" <a href=\""+ valid +"\">valid</a><br>");
  ```

Enquanto estas sugestões irão diminuir sua exposição dramaticamente, ataques CSRF avançados podem burlar muitas dessas restrições. As melhores técnicas são o uso de tokens únicos e a eliminação de todas as vulnerabilidades XSS na sua aplicação.

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0192
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5116
- MySpace Samy Interview: http://blog.outer-court.com/archive/2005-10-14-n81.html
- An attack which uses Quicktime to perform CSRF attacks http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9005607&intsrc=hm_list

REFERÊNCIAS

- CWE: CWE-352 (Cross-Site Request Forgery)
- WASC Threat Classification: No direct mapping, but the following is a close match:
- http://www.webappsec.org/projects/threat/classes/abuse_of_functionality.shtml
- OWASP CSRF, http://www.owasp.org/index.php/Cross-Site_Request_Forgery
- OWASP Testing Guide, https://www.owasp.org/index.php/Testing_for_CSRF
- OWASP CSRF Guard, http://www.owasp.org/index.php/CSRF_Guard
- RSnake, "What is CSRF?", http://ha.ckers.org/blog/20061030/what-is-csrf/
- Jeremiah Grossman, slides and demos of “Hacking Intranet sites from the outside”
- http://www.whitehatsec.com/presentations/whitehat_bh_pres_08032006.tar.gz
- HTTP Data Validation Framework, http://www.hdiv.org
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A6 – FALHAS DE CONFIGURAÇÃO DE SEGURANÇA

Aplicações podem acidentalmente deixar vazar informações sobre sua configuração, rotinas internas ou cometer violações de privacidade devido a uma variedade de problemas. Elas também podem vazar informações sobre suas rotinas de processamento de estado através do tempo que demoram em processar requisições específicas ou como reagem (response) a diferentes entradas de informação (request), como exibir o mesmo texto de erro, mas com números diferentes. Aplicações web irão frequentemente vazar informações do seu estado interno através de mensagens de erro detalhadas. Estas informações poderão ser utilizadas lançar ou automatizar ataques mais poderosos.

AMBIENTES AFETADOS

Todos os frameworks de aplicações web estão vulneráveis a falhas de configuração de segurança.

VULNERABILIDADES

Aplicações geralmente apresentam mensagens de erro para seus usuários. Frequentemente estas mensagens são úteis para um atacante, já que elas revelam detalhes de implementação ou informações que são úteis para a exploração de uma vulnerabilidade específica. Os exemplos mais comuns são:

- Tratamento de erro detalhado, onde a indução de um erro causa a exibição de muita informação, como falhas de *queries* SQL ou outras informações de *debug*.
- Funções que produzem resultados diferentes dependendo da entrada de dados. Por exemplo: uma função de autenticação deveria produzir exatamente a mesma mensagem de erro para os casos em que o usuário não existe e para quando a senha está incorreta.

VERIFICANDO A SEGURANÇA

O objetivo é verificar se a aplicação não vaza informações via mensagens de erro ou outros meios.

Abordagem automatizada: Ferramentas de scanner vulnerabilidades irão normalmente causar a geração de mensagens de erro. Ferramentas de análise estática podem procurar pelo uso de APIs que vazam informações, mas não serão capazes de verificar o significado dessas mensagens.

Abordagem manual: Uma revisão de código pode procurar por tratamento inapropriado de erros e outros padrões que vazam informações, mas isto consome muito tempo. Testes irão gerar mensagens de erro, mas saber quais caminhos de erros foram cobertos é um desafio.

PROTEÇÃO

Desenvolvedores devem utilizar ferramentas como o WebScarab ou o ZAP da OWASP para tentar fazer suas aplicações gerarem erros. Aplicações que não tenham sido testadas desta maneira certamente irão gerar erros inesperados. Aplicações devem também incluir uma arquitetura padrão de tratamento de exceções para prevenir que informações vazem para os atacantes.

Prevenir o vazamento de informações requer disciplina. As seguintes práticas se provaram efetivas:

- Certifique-se de que todo o time de desenvolvimento do software compartilhe uma abordagem em comum para o tratamento de exceções.
- Desabilite ou limite tratamento de erros detalhados. Em particular, não exiba informações de debug para os usuários finais, stack traces (informações da pilha) ou informações de caminhos.
- Várias camadas podem retornar resultados ou exceções fatais, como a camada de banco de dados e servidores web. É vital que os erros de todas as camadas sejam checados adequadamente e configurados para esconder estes detalhes do atacante.
- Tenha cuidado com frameworks que retornam códigos de erros HTTP diferentes dependendo se o erro esta dentro do seu código ou dentro do código do framework. Vale a pena criar um tratador de erros padrão que retorne uma mensagem de erro sanitizada apropriadamente para a maioria dos usuários em produção.
- Sobrescrever o tratador de erro padrão para que ele sempre retorne páginas de erro “200” (OK) reduz a probabilidade de ferramentas de scanning automatizadas de determinarem se o erro sério ocorreu. Mesmo sendo isso uma “segurança por obscuridade”, pode fornecer uma camada extra de defesa.
- Algumas empresas maiores decidiram incluir códigos de erro aleatórios ou únicos dentro de suas aplicações. Isto pode ajudar o help desk em encontrar a solução correta para um erro em particular, mas também pode permitir que atacantes determinem onde e quando a aplicação falhou.
- Sempre dê a mensagem de erro que “O usuário ou a senha não estão corretos” do que “A senha não está correta” para falhas de login.
- Certifique-se de que a aplicação sempre retorne o código HTTP 200 ou 302 no ocorrência de um erro.
- Utilize as classes EnterpriseSecurityException e HTTPUtils da OWASP ESAPI:

```
} catch (EnterpriseSecurityException e) {
   auth.getCurrentUser().logout(request,response);
   RequestDispatcher dispatcher = request.getRequestDispatcher("WEB-
INF/admin/login.jsp");
   dispatcher.forward(request, response);
}
```

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4899
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3389
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0580

REFERÊNCIAS

- CWE: CWE-200 (Information Leak), CWE-203 (Discrepancy Information Leak), CWE-215 (Information Leak Through Debug Information), CWE-209 (Error Message Information Leak), others.
- WASC Threat Classification: http://www.webappsec.org/projects/threat/classes/information_leakage.shtml
- OWASP, http://www.owasp.org/index.php/Error_Handling
- OWASP, http://www.owasp.org/index.php/Category:Sensitive_Data_Protection_Vulnerability
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A7 – ARMAZENAMENTO CRIPTOGRÁFICO INSEGURO

Proteger dados sensíveis com criptografia tem se tornado uma parte importante na maioria das aplicações web. Falhas ao criptografar os dados sensíveis são bastante difundidas. Aplicações que criptografam frequentemente contêm criptografia desenvolvida de forma errada (ou fraca) utilizando cifras inapropriadas ou cometendo sérios erros utilizando cifras fortes. Estas falhas podem levar ao comprometimento de dados sensíveis e violações de conformidade.

AMBIENTES AFETADOS

Todos os frameworks de aplicações Java EE estão vulneráveis ao armazenamento criptográfico inseguro.

VULNERABILIDADES

Prevenir falhas de criptografia exige um planejamento cuidadoso. Os problemas mais comuns são:

- Não criptografar dados sensíveis
- Utilizar algoritmos locais
- Uso inseguro de algoritmos fortes
- Uso continuado de algoritmos comprovadamente fracos (MD5, SHA-1, RC3, RC4, etc.)
- Chaves codificadas permanentemente ou armazenando as chaves em locais desprotegidos

VERIFICANDO A SEGURANÇA

O objetivo é verificar se a aplicação criptografa apropriadamente informações sensíveis no armazenamento.

Abordagem automatizada: Ferramentas de análise de vulnerabilidades não podem verificar o armazenamento criptográfico de forma alguma. Ferramentas de análise de código podem detectar o uso de APIs de criptografia conhecidas, mas não podem detectar se elas estão sendo utilizadas de forma correta ou se a criptografia é realizada em um componente externo.

Abordagem manual: Assim como a análise, testes não podem verificar o armazenamento criptográfico. Revisão de código é a melhor maneira de verificar se uma aplicação criptografa dados sensíveis e tem o mecanismo e a gerência das chaves implementados de forma apropriada. Isto pode envolver a auditoria e revisão da configuração de sistemas externos em alguns casos.

PROTEÇÃO

O aspecto mais importante é garantir que tudo que deveria ser criptografado está realmente criptografado e protegido. Então você deve garantir que a criptografia está sendo utilizada apropriadamente. Como existem muitas maneiras de utilizar a criptografia de forma inapropriada, as seguintes recomendações devem ser utilizadas como parte da sua rotina de testes para ajudar a garantir a manipulação segura de materiais criptografados.

- Não crie algoritmos criptográficos. Utilize apenas algoritmos públicos e aprovados como AES, RSA, Criptografia de chave pública e SHA-256 ou maior para hashing.
- Não use algoritmos fracos como MD5 ou SHA-1. Prefica alternativas mais seguras como SHA-256 ou maior.
- Gere as chaves offline e armazene as chaves privadas com extrema cautela. Nunca transmita chaves privadas por um canal inseguro.
- Certifique-se de que as credenciais de infra-estrutura como banco de dados, middlewares, etc, estão seguras (via permissões e controles do sistema de arquivos) ou criptografadas de forma segura e não são facilmente descriptografados por usuários locais ou remotos.
- Hashing não é criptografia. Se um atacante sabe qual o algoritmo de hash que está sendo utilizando, ele pode realizar um ataque de força bruta para quebrar o hash, ou seja, advinhar qual palavra gerou aquele hash.
- Certifique-se de que os dados criptografados armazenados no disco não são fáceis de serem descriptografados. Por exemplo, criptografia de banco dados é inútil se o pool de conexões do banco prover acesso sem criptografia
- No requisito 3 do PCI Data Security Standard (DSS), você deve proteger os dados do cartão crédito. Conformidade com o PCI DSS é obrigatória desde 2008 para todos os comerciantes ou qualquer outro que manipule cartões de crédito. Uma boa prática é nunca armazenar dados desnecessários, como as informações da fita magnética ou o número da conta primária (PAN em inglês, também conhecido como número do cartão de crédito). Se você armazena este número, os requisitos de conformidade com o DSS são significativos. Por exemplo, você nunca deve armazenar o número de verificação ou CVV (um número de três dígitos na parte de trás do cartão) sob qualquer circunstância. Para mais informações, veja o Guia do PCI DSS.
- Utilize a classe Encryptor da OWASP ESAPI.

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6145
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1664
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-1101 (Verdade para a maioria dos containers Java EE também)

REFERÊNCIAS

- CWE: CWE-311 (Failure to encrypt data), CWE-326 (Weak Encryption), CWE-321 (Use of hard-coded cryptographic key), CWE-325 (Missing Required Cryptographic Step), others.
- WASC Threat Classification: No explicit mapping
- OWASP, http://www.owasp.org/index.php/Cryptography
- OWASP Guide, http://www.owasp.org/index.php/Guide_to_Cryptography
- OWASP, http://www.owasp.org/index.php/Insecure_Storage
- OWASP, http://www.owasp.org/index.php/How_to_protect_sensitive_data_in_URL's
- PCI Data Security Standard, https://www.pcisecuritystandards.org
- Bruce Schneier, http://www.schneier.com/
- Bouncy Castle Crypto APIs for Java, www.bouncycastle.org
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A8 – FALHA NA RESTRIÇÃO DE ACESSO A URLS

Frequentemente, a única proteção para uma URL é que links para aquela página não são apresentados para usuários não autorizados. Entretanto, um atacante motivado, habilidoso ou apenas sortudo pode ser capaz de descobrir e acessar estas páginas, chamando funções e visualizando dados. Esta segurança por obscuridade não é suficiente para proteger funções sensíveis e dados em uma aplicação. Verificação de controle de acesso deve ser realizada antes que um pedido para funções sensíveis seja realizado, o que garante que o usuário está autorizado a acessar aquela função.

AMBIENTES AFETADOS

Todos os frameworks de aplicações Java EE estão vulneráveis a falhas de restrição de acesso a URLs

VULNERABILIDADES

O método primário de ataque para esta vulnerabilidade é chamado de “forced browsing” (ou navegação forçada, em português), que envolve adivinhar links e utilizar força bruta para encontrar páginas desprotegidas. A ferramenta para realizar esta tarefa é o DirBuster da OWASP (veja as referências).

Aplicações frequentemente permitem que o código de controle de acesso evolua e se espalhe por toda base de código, resultando num modelo complexo que é difícil de ser entendido por desenvolvedores e especialistas em segurança. Esta complexidade aumenta a probabilidade de que erros irão acontecer e de que páginas serão esquecidas, deixando-as expostas.

Alguns exemplos dessas falhas incluem:

- URLs “escondidas” ou “especiais”, exibidas apenas para administrador ou usuários privilegiados na camada de apresentação, mas acessíveis a todos os usuários se eles souberem que ela existe, como /admin/adduser.php ou /approveTransfer.do
- Páginas utilizadas durante o desenvolvimento ou teste que são páginas modelo para autorização de perfis e são enviadas para o ambiente de produção.
- Muitas vezes aplicações permitem acesso a arquivos “ocultos”, como XML estático ou relatórios gerados pelo sistema, confiando na segurança por obscuridade para escondê-los.
- Código que força a política de controle de acesso, mas está desatualizado ou é insuficiente. Por exemplo, imagine /approveTransfer.do era disponível para todos os usuários, mas desde os controles da SOX foram implementados, ela só deve estar disponível para os “aprovadores”. O ajuste pode ter sido para

não mostrá-la para usuários não autorizados, mas nenhum controle de acesso é utilizado quando a página é solicitada.

- Código que avalia os privilégios no cliente mas não no servidor, como este ataque no MacWorld 2007, que aprovou entradas "Platinum" avaliadas em $1700 via JavaScript no navegador e não no servidor.

VERIFICANDO A SEGURANÇA

O objetivo é verificar se o controle de acesso é verificado de forma consistente na camada de apresentação e na lógica do negócio para todas as URLs da aplicação.

Abordagem automatizada: Scanners de vulnerabilidades e ferramentas de análise estática tem dificuldade em verificar o controle de acesso na URL, mas por razões diferentes. Scanners de vulnerabilidades têm dificuldade em adivinhar páginas ocultas e determinar que páginas devam ser permitidas para cada usuário, enquanto a análise estática tem problemas para identificar controles de acesso customizados no código e ligar a camada de apresentação com a lógica de negócio.

Abordagem manual: A abordagem mais eficiente e precisa é utilizar um combinação de revisão de código e testes de segurança para verificar o mecanismo de controle de acesso. Se o mecanismo for centralizado, a verificação pode ser bastante eficiente. Se o mecanismo for distribuído por toda a base de código, a verificação pode ser mais demorada. Se o mecanismo for executado externamente, a configuração deve ser examinada e testada.

PROTEÇÃO

Planejar a autorização através da criação de uma matriz para mapear os perfis e funções da aplicação é um passo chave para obter proteção contra acesso irrestrito a URLs. Aplicações web devem executar o controle de acesso em todas as URLs e funções de negócio. Não é suficiente colocar o controle de acesso na camada de apresentação e deixar a lógica de negócio desprotegida. Também não é suficiente verificar apenas uma vez durante o processo para garantir que o usuário está autorizado e então não checar novamente em passos subseqüentes. Caso contrário, um atacante pode simplesmente pular o passo onde a autorização é verificada e forjar os valores dos parâmetros necessários para continuar no próximo passo.

Habilitar o controle de acesso de URL requer um planejamento cuidadoso. Entre as mais importantes considerações estão:

- Garantir que a matriz de controle de acesso faz parte do negócio, arquitetura e design da aplicação
- Garantir que todas as URLs e funções de negócio estão protegidas por um controle de acesso efetivo que verifica o perfil do usuário e suas permissões antes de realizar qualquer processamento. Certifique-se de que isto é feito durante todos os passos, não apenas uma vez no começo de qualquer processo com múltiplos passos. Isto pode ser configurado no web.xml utilizando

security-constraint e auth-constraint para permitir acesso a perfis Java EE para a URL dessa forma:

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>
                Páginas de administração da Aplicação Java EE protegidas
                </web-resource-name>
                <description>Requer autenticação dos usuários.</description>
                <url-pattern>/admin/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
                <description>
                Permite acesso às paginas de administração
                </description>
                <role-name>Administrador</role-name>
        </auth-constraint>
</security-constraint>
<security-role>
        <description>Administrador Java EE</description>
        <role-name>Administrador</role-name>
</security-role>
```

Outra abordagem é utilizar o Spring Security, um framework Java EE de segurança para autenticação e autorização.

- Realize um teste de invasão antes entregar o código ou colocá-lo em produção para garantir que a aplicação não possa ser utilizada de forma incorreta por um atacante motivado e habilidoso.
- Preste atenção em arquivos de inclusão e biblioteca, especialmente se eles possuírem extensões executáveis como .php. Quando possível, estes devem ser mantidos fora do web root. Eles devem verificar se não estão sendo acessados diretamente, por exemplo, verificar por uma constante que
  só pode ser criada pela biblioteca
- Não presuma que usuários não terão conhecimento de URLs ou APIs ocultas ou especiais. Sempre garanta que ações administrativas e de alto privilegio estão protegidas.
- Bloqueie o acesso a todos os tipos de arquivos que sua aplicação nunca deverá fornecer. Idealmente, este filtro deve seguir a abordagem whitelist e apenas permitir tipos de arquivos que você pretende fornecer como por exemplo .html, .pdf, .php. Isto iria então bloquear qualquer tentativa de acessas arquivos de log, arquivos XML, etc. que você nunca forneceria diretamente.
- Estabeleça uma política de segurança e habilite o Gerenciador de Segurança do Java (Java Security Manager).
- Mantenha seu antivírus e o sistema sempre atualizados para componentes como processadores de XML, arquivos texto e de imagem, etc, que manipulam dados informados pelos usuários.

- O HTTP Data Integrity Validator permite acesso apenas para URLs que foram retornadas para o usuário. Isto significa que ataques de força bruta não irão funcionar e que verificações de autorização adicionais são implementadas para um perfil do usuário. Um usuário comum pode digitar a URL manualmente, mas HDIV não permitirá acesso.
- Utilize a classe AcessController da OWASP Enterprise Security API.

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0147
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0131
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1227

REFERÊNCIAS

- CWE: CWE-325 (Direct Request), CWE-288 (Authentication Bypass by Alternate Path), CWE-285 (Missing or Inconsistent Access Control)
- WASC Threat Classification: http://www.webappsec.org/projects/threat/classes/predictable_resource_location.shtml
- OWASP, http://www.owasp.org/index.php/Forced_browsing
- OWASP Guide, http://www.owasp.org/index.php/Guide_to_Authorization
- HTTP Data Validation Framework, http://www.hdiv.org
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI
- DirBuster, https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project
- Spring Security, http://static.springsource.org/spring-security/site/index.html

A9 – PROTEÇÃO INSUFICIENTE NA CAMADA DE TRANSPORTE

As aplicações frequentemente falham em criptografar o tráfego da rede quando é necessário proteger comunicações sensíveis. Criptografia (normalmente SSL) deve ser utilizada para todas as conexões autenticadas, especialmente páginas web acessíveis pela Internet, mas conexões de backend também. Caso contrário, a aplicação irá expor uma autenticação ou token de sessão. Adicionalmente, criptografia deve ser empregada sempre que dados sensíveis, como cartões de crédito ou informações de saúde são transmitidas. As aplicações que saem ou podem ser forçadas a sair do modo de criptografia podem ser manipuladas por atacantes. O padrão PCI requer que todas as informações de cartão de crédito sejam transmitidas através da internet criptografadas.

AMBIENTES AFETADOS

Todos os frameworks de aplicações Java EE são vulneráveis à Proteção Insuficiente na Camada de Transporte

VULNERABILIDADES

Falha ao criptografar comunicações sensíveis significa que um atacante que pode capturar tráfego da rede poderá acessar a conversa, incluindo quaisquer credenciais ou informações sensíveis transmitidas. Considere que redes diferentes serão mais ou menos suscetíveis a captura de pacotes (sniffing). Entretanto, é importante perceber que eventualmente um host será comprometido em quase todas as redes e atacantes irão rapidamente instalar um sniffer para capturar as credenciais de outros sistemas.

Utilizar SSL para comunicações com usuários finais é crítico, pois eles muito provavelmente estarão utilizando redes inseguras para acessar as aplicações. Porque o HTTP inclui as credenciais de autenticação ou um token de sessão em todos os requests (exceto para cookies com o atributo secure), todo o tráfego autenticado precisa ser por SSL, não apenas o pedido de login.

Criptografar as comunicações com os servidores de backend também é importante. Entretanto estas redes provavelmente sejam mais protegidas, as informações e as credenciais que carregam são mais sensíveis e mais amplas. Portanto, usar SSL no backend é muito importante.

Criptografar dados sensíveis, como cartões de créditos e CPFs, tem se tornado um regulamento de privacidade e financeiro para muitas organizações. Negligenciar o uso de SSL para conexões que manipulam estes dados cria um risco de conformidade.

VERIFICANDO A SEGURANÇA

O objetivo é verificar se a aplicação criptografa todas as comunicações autenticadas e sensíveis de forma apropriada.

Abordagem automatizada: Ferramentas de análise de vulnerabilidades podem verificar se o SSL está sendo utilizando no front end e pode encontrar diversas falhas relacionadas a ele. Entretanto estas ferramentas não têm acesso às conexões de back end e não podem verificar se elas são seguras. Ferramentas de análise estática podem ajudar com a análise de chamadas para os sistemas de back end, mas provavelmente não irão entender a lógica exigida para todos os tipos de sistemas.

Abordagem manual: Testes podem verificar se SSL está sendo utilizado e encontrar muitas falhas relacionadas no front end, mas abordagens automatizadas são provavelmente mais eficientes. Revisão de código também é eficiente para verificar o uso apropriado de SSL para todas as conexões de back end.

PROTEÇÃO

A proteção mais importante é utilizar SSL em qualquer conexão autenticada sempre que dados sensíveis estão sendo transmitidos. Existem diversos detalhes envolvidos em configurar o SSL para aplicações web de forma apropriada, então é importante entender e analisar o seu ambiente. Por exemplo, IE 7.0 coloca uma barra verde para certificados de alta confiança, mas isto apenas não é um controle apropriado para provar o uso seguro de criptografia.

- Utilize SSL para todas as conexões que são autenticadas ou transmitem dados sensíveis ou de valor, como credenciais, detalhes de cartão de crédito, informações de saúde e outras informações privadas.
- Certifique-se de que as comunicações entre elementos de infraestrutura, como entre o servidor web e o sistema de banco de dados, estão protegidas apropriadamente através do uso de segurança na camada de transporte ou criptografia a nível de protocolo para credenciais e dados valiosos.
- Proteja o cookie de sessão setando o secure bit para 1 (javax.servlet.http.Cookie.setSecure(true)). Isto irá prevenir o envio do cookie em texto plano.
- Quando utilizar SSL, utilize para toda a sessão. Apenas proteger as credenciais de login é insuficiente porque dados e informações da sessão devem ser criptografados também.
- No requisito 4 do PCI Data Security Standard, você deve proteger dados de cartão de crédito em trânsito. Conformidade com o PCI DSS é obrigatória desde 2008 para todos os comerciantes ou qualquer outro que manipule cartões de crédito. Em geral, cliente, parceiro e acesso administrativo online aos sistemas devem ser criptografados utilizando SSL ou similar. Para mais informações, veja o Guia do PCI DSS.
- Adicione uma restrição de segurança no web.xml para todas as URLs que necessitem de HTTPS

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>Páginas HTTPS</web-resource-name>
                <url-pattern>/profile</url-pattern>
                <url-pattern>/register</url-pattern>
```

```
            <url-pattern>/password-login</url-pattern>
            <url-pattern>/ldap-login</url-pattern>
        </web-resource-collection>
        <user-data-constraint>
            <transport-guarantee>CONFIDENCIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
```

- Use a classe HTTPUtilities da OWASP ESAPI para criar um cookie seguro.

EXEMPLOS

- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6430
- http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-4704
- http://www.schneier.com/blog/archives/2005/10/scandinavian_at_1.html

REFERÊNCIAS

- CWE: CWE-311 (Failure to encrypt data), CWE-326 (Weak Encryption), CWE-321 (Use of hard-coded cryptographic key), CWE-325 (Missing Required Cryptographic Step), others.
- WASC Threat Classification: No explicit mapping
- OWASP Testing Guide, Testing for SSL / TLS, https://www.owasp.org/index.php/Testing_for_SSL-TLS
- OWASP Guide, http://www.owasp.org/index.php/Guide_to_Cryptography
- Foundstone - SSL Digger, http://www.foundstone.com/index.htm?subnav=services/navigation.htm&subcontent=/services/overview_s3i_des.htm
- NIST, SP 800-52 Guidelines for the selection and use of transport layer security (TLS) Implementations, http://csrc.nist.gov/publications/nistpubs/800-52/SP800-52.pdf
- NIST SP 800-95 Guide to secure web services, http://csrc.nist.gov/publications/drafts.html#sp800-95
- OWASP Enterprise Security API - http://www.owasp.org/index.php/ESAPI

A10 – REDIRECTS E FORWARDS NÃO VALIDADOS

É a não proteção de URLs (de certa forma relacionada ao acesso inseguro de URLs) que são acessadas por redirects e forwards.

AMBIENTES AFETADOS

Todos os frameworks de aplicações Java EE são vulneráveis a redirects e forwards não validados.

VULNERABILIDADES

Redirects e Forwards são extremamente comuns em aplicações web. Os problemas começam a aparecer em duas ocasiões:

- Quando parametrizamos a URL para qual o usuário será redirecionado/encaminhado, sem a presença de uma rotina de validação.
- Quando não validamos a ocasião em que o usuário será redirecionado/encaminhado, permitindo a exploração destes casos.

PROTEÇÃO

- Analise de onde o usuário está vindo, de forma a mapear a aplicação e impedir acessos indevidos a URLs
- Valide qualquer caso de redirect ou forward parametrizável
- Preste especial atenção aos casos parametrizados que precisam, necessariamente, utilizar URLs externas.

EXEMPLOS

- http://www.example.com/redirect.jsp?url=evil.com
- http://www.example.com/boring.jsp?fwd=admin.jsp

REFERÊNCIAS

- Open Redirects, https://www.owasp.org/index.php/Open_redirect
- Método SecurityWrapperResponsesendRedirect() da ESAPI, http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/filters/SecurityWrapperResponse.html#sendRedirect(java.lang.String)
- CWE Entry 601 on Open Redirects, http://cwe.mitre.org/data/definitions/601.html
- WASC Article on URL Redirector Abuse, http://projects.webappsec.org/URL-Redirector-Abuse
- Google blog article on the dangers of open redirects, http://googlewebmastercentral.blogspot.com/2009/01/open-redirect-urls-is-your-site-being.html

PARA ONDE IR?

O OWASP Top 10 é apenas o começo da sua jornada em segurança de aplicações web.

> The world's six billion people can be divided into two groups: group one, who knows why every good software company ships products with known bugs; and group two, who don't. Those in group 1 end to forget what life was like before our youthful optimism was spoiled by reality. Sometimes we encounter a person in group two …who is shocked that any software company would ship a product before every last bug is fixed.
>
> Eric Sink, Guardian May 25, 2006

A maioria dos nossos clientes e usuários está no grupo dois. Como você lida com este problema é uma oportunidade para melhorar seu código e o estado de segurança da sua aplicação web. Bilhões de dólares são gastos todo ano e muitos milhões de pessoas sofrem roubo de identidade e fraudes devido a vulnerabilidades discutidas neste documento.

Para arquitetos e engenheiros (de software)

Para proteger suas aplicações apropriadamente, você deve saber o que você está protegendo (classificação dos ativos), saber das ameaças e os riscos de insegurança e abordá-los de maneira estruturada. Projetar qualquer aplicação não trivial requer uma boa dose de segurança.

- Certifique-se de que você utiliza segurança “suficiente” baseada na modelagem de riscos e classificação dos ativos. Entretanto, como as leis de conformidade (SOX, HIPAA, Basel, etc) impõe altas penalidades, é apropriado investir mais tempo e recursos além do mínimo exigido hoje em dia, particularmente se melhores práticas são conhecidas e consideravelmente melhores que o mínimo.
- Pergunte sobre os requisitos do negócio, particularmente requisitos não-funcionais que estejam faltando
- Trabalhe através do OWASP Secure Software Contract Annex com o seu cliente.
- Garanta ter considerado confidencialidade, integridade, disponibilidade e não-repúdio
- Certifique-se de que seus projetos são consistentes com as políticas de segurança e os padrões como COBIT ou PCI DSS

Para desenvolvedores

Muitos desenvolvedores já possuem um bom entendimento básico sobre segurança em aplicações web. Para garantir o domínio total na segurança em aplicações web

requer prática. Qualquer um pode destruir (ex. realizar testes de invasão), mas é necessário um mestre para construir software seguro. Torne-se um mestre.

- Considere participar da OWASP e ir às reuniões dos capítulos locais.
- Solicite treinamento em desenvolvimento seguro se você tiver recursos para treinamentos. Solicite recursos se não tiver.
- Desenvolva suas funcionalidades de forma segura
- Adote padrões de codificação que auxiliam na construção de códigos mais seguros
- Refatore códigos existentes para utilizar construtores mais seguros na plataforma escolhida, como queries parametrizadas
- Leia o OWASP Guide e comece a aplicar os controles ao seu código. Ao contrário de muitos guias de segurança, ele é desenvolvido para ajudá-lo a construir software seguro, não quebrá-lo.
- Teste seu código por defeitos de segurança e faça disto parte do seu procedimento de testes unitários e web
- Veja as referências de livros e verifique se algum deles é aplicável para o seu ambiente.

Para projetos open source

Projetos open source são um desafio particular para a segurança em aplicações web. Existem literalmente milhões de projetos de código aberto, desde um projeto pessoal de apenas um desenvolvedor a projetos maiores como o Apache, Tomcat e aplicações web de larga escala como Drupal.

- Considere participar da OWASP e ir às reuniões dos capítulos locais.
- Se o seu projeto tiver mais do que 4 desenvolvedores, eleja pelo menos um deles para se encarregar da segurança
- Adote padrões de codificação que auxiliam na construção de códigos mais seguros
- Adote a política de *responsible disclosure* para garantir que os defeitos de segurança são tratados apropriadamente
- Veja as referências de livros e verifique se algum deles é aplicável para o seu ambiente

Para proprietários de aplicações

Proprietários de aplicações comerciais são, normalmente, restritos de recursos e tempo. Por isso, eles deveriam:

- Trabalhar através do OWAP Secure Software Contract Annex com os desenvolvedores
- Certifique-se de que os requisitos de negócio incluem requisitos não-funcionais como os requisitos de segurança

- Contrate ou treine desenvolvedores que possuem conhecimentos em segurança
- Teste por defeitos de segurança durante todo o projeto: planejamento, implementação, teste e implantação
- Possua recursos, orçamento e tempo no plano do projeto para remediar problemas de segurança

Para os executivos e gerentes

Sua organização deve ter um ciclo de vida de desenvolvimento seguro (SDLC, em inglês) em prática que sirva para sua empresa. Vulnerabilidades são muito mais baratas de corrigir durante o desenvolvimento do que depois que o seu produto é implantado. Um SDLC razoável não inclui apenas testes para o Top 10, ele inclui também:

- Para softwares de prateleira, certifique-se de comprar políticas e contratos que incluem requisitos de segurança
- Para códigos customizados, adote os princípios de programação segura nas suas políticas e padrões
- Treine seus desenvolvedores em técnicas de desenvolvimento seguro e garanta que eles mantenham estas habilidades atualizadas
- Inclua ferramentas de análise de segurança de código no seu orçamento
- Notifique seus desenvolvedores de software da importância da segurança
- Treine seus arquitetos, projetistas e as pessoas de negócio em fundamentos de segurança em aplicações web
- Considere contratar auditores de código para realizarem uma avaliação independente
- Adote práticas de *responsible disclosure* e construa um processo para responder apropriadamente os relatórios de vulnerabilidades dos seus produtos

REFERÊNCIAS

Projetos da OWASP

OWASP é o primeiro site para segurança em aplicações web. O site da OWASP contém diversos projetos, fóruns, blogs, apresentações, ferramentas e artigos. A OWASP realiza quatro grandes conferências sobre segurança em aplicações web por ano, nos quatro cantos do mundo, e tem centenas capítulos locais. Os seguintes projetos provavelmente serão úteis:

- OWASP Enterprise Security API project (ESAPI)
- OWASP Guide to Building Secure Web Applications
- OWASP Testing Guide
- OWASP Code Review Project
- OWASP Java Project
- OWASP .NET Project

Livros

Esta não é uma lista exaustiva. Use estas referências para encontrar os livros que sirvam para as suas necessidades:

- [ALS1] Alshanetsky, I. “php|architect's Guide to PHP Security”, ISBN 0973862106
- [BAI1] Baier, D., “Developing more secure ASP.NET 2.0 Applications”, ISBN 978-0-7356-2331-6
- [GAL1] Gallagher T., Landauer L., Jeffries B., "Hunting Security Bugs", Microsoft Press, ISBN 073562187X
- [GRO1] Fogie, Grossman, Hansen, Rager, “Cross Site Scripting Attacks: XSS Exploits and Defense”, ISBN 1597491543
- [HOW1] Howard M., Lipner S., "The Security Development Lifecycle", Microsoft Press, ISBN 0735622140
- [SCH1] Schneier B., "Practical Cryptography", Wiley, ISBN 047122894X
- [SHI1] Shiflett, C., “Essential PHP Security”, ISBN 059600656X
- [WYS1] Wysopal et al, The Art of Software Security Testing: Identifying Software Security Flaws, ISBN 0321304861

Web Sites

- OWASP, http://www.owasp.org
- MITRE, Common Weakness Enumeration – Vulnerability Trends, http://cwe.mitre.org/documents/vulntrends.
- html
- Web Application Security Consortium, http://www.webappsec.org/
- SANS Top 20, http://www.sans.org/top20/
- PCI Security Standards Council, publishers of the PCI standards, relevant to all organizations processing or

- holding credit card data, https://www.pcisecuritystandards.org/
- Build Security In, US CERT, https://buildsecurityin.us-cert.gov/daisy/bsi/home.html