
Semana 20 - Aula 1

Tópico Principal da Aula: Servidores web, API e segurança das aplicações

Subtítulo/Tema Específico: Testes automatizados de segurança

Código da aula: [SIS] ANO2C2B3S20A1

Objetivos da Aula:

- Implementar testes automatizados para identificar vulnerabilidades de segurança.
- Ter persistência ao realizar testes contínuos para garantir a segurança.

Recursos Adicionais:

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.

Exposição do Conteúdo:

Referência do Slide: Slide 06 - Ponto de partida

- **Definição:** A segurança em sistemas financeiros digitais é uma prioridade absoluta. Uma *fintech*, por exemplo, que processa pagamentos online, precisa garantir que suas APIs (Interfaces de Programação de Aplicações) sejam robustas contra ataques. Vulnerabilidades como injeções de SQL podem permitir que invasores acessem ou manipulem o banco de dados, enquanto falhas na segurança podem expor dados sensíveis de clientes, como informações de cartão de crédito.
- **Aprofundamento/Complemento:** **A identificação de vulnerabilidades antes que a aplicação seja lançada (em produção) é uma prática essencial conhecida como "Shift-Left Security". Isso significa integrar a segurança no início do ciclo de desenvolvimento, em vez de tratá-la como uma etapa final. Em arquiteturas *serverless* (sem servidor), onde a aplicação é composta por várias funções e serviços de terceiros, os testes são ainda mais críticos, pois a superfície de ataque é distribuída e complexa.**
- **Exemplo Prático:** Imagine que um desenvolvedor cria uma API de pagamento. Sem testes de segurança, ele pode não perceber que uma consulta ao banco de dados para verificar o status de um pagamento pode ser manipulada por um atacante para extrair dados de todas as transações, em vez de apenas uma.

Referência do Slide: Slide 07 a 11 - Ferramentas de testes automatizados de segurança

- **Definição:** **Testes automatizados de segurança são processos que utilizam ferramentas especializadas para escanear, analisar e identificar falhas de segurança em uma aplicação de forma contínua e escalável**. Essas ferramentas simulam ataques e mapeiam pontos fracos, permitindo que as equipes de desenvolvimento corrijam as falhas proativamente.

- **Aprofundamento/Complemento: A automação é fundamental na cultura DevOps/DevSecOps, pois permite que os testes de segurança sejam executados a cada nova alteração no código, garantindo que novas vulnerabilidades não sejam introduzidas. As principais ferramentas apresentadas são:**
    1. **OWASP ZAP (Zed Attack Proxy):**
        - **Definição:** É uma ferramenta gratuita e de código aberto, mantida pela OWASP (Open Web Application Security Project), que atua como um "proxy de ataque". Ela se posiciona entre o navegador do testador e a aplicação web para interceptar e modificar o tráfego, ajudando a encontrar vulnerabilidades como injeções de SQL, Cross-Site Scripting (XSS) e configurações de segurança inadequadas.
        - **Exemplo Prático:** Um testador pode usar o OWASP ZAP para escanear automaticamente uma API de pagamento. A ferramenta enviará uma série de requisições maliciosas para os *endpoints* (pontos de extremidade) da API e, ao final, gerará um relatório indicando se algum *endpoint* está, por exemplo, expondo dados de cartão de crédito em sua resposta sem a devida criptografia.
        - **Vídeo Sugerido:** OWASP ZAP - Primeiros Passos
    2. **Burp Suite:**
        - **Definição:** É uma plataforma robusta (com versões gratuita e paga) para realizar testes de segurança em aplicações web. Sua principal função é interceptar requisições HTTP, permitindo que o testador as analise e modifique manualmente. Também oferece módulos para automação de ataques.
        - **Exemplo Prático:** Para testar uma vulnerabilidade de Cross-Site Scripting (XSS), um analista pode usar o Burp Suite para interceptar uma requisição de login e inserir um script malicioso no campo de nome de usuário. Se a aplicação não validar essa entrada corretamente, o script pode ser executado no navegador de um administrador, roubando suas credenciais.
        - **Vídeo Sugerido:** O que é Burp Suite?
    3. **Nmap (Network Mapper):**
        - **Definição:** É uma ferramenta de código aberto utilizada para exploração de redes e auditoria de segurança. Ela permite descobrir quais servidores estão ativos em uma rede, quais serviços (aplicações) eles oferecem e quais portas estão abertas, identificando potenciais pontos de entrada para um ataque.
        - **Exemplo Prático:** Um administrador de sistemas pode usar o Nmap para escanear os servidores da empresa e verificar se alguma porta que deveria ser privada (como a porta de acesso ao banco de dados) está acidentalmente exposta à internet.
        - **Vídeo Sugerido:** Introdução ao Nmap
    4. **Snyk:**

        - **Definição:** É uma plataforma de segurança focada em encontrar e corrigir vulnerabilidades nas dependências de código aberto (bibliotecas e *frameworks*) que sua aplicação utiliza.
        - **Exemplo Prático:** Ao desenvolver um sistema de pagamentos, sua equipe pode usar uma biblioteca de criptografia de terceiros. O Snyk pode escanear o projeto, identificar essa biblioteca, verificar se ela possui alguma vulnerabilidade de segurança conhecida e sugerir a atualização para uma versão mais segura.
    - **Vídeo Sugerido:** O que é o Snyk? Análise de Vulnerabilidades

---

Referência do Slide: Slide 12 e 13 - Análise de vulnerabilidades

- **Definição:** **A análise de vulnerabilidades é o processo de identificar, classificar, priorizar e corrigir riscos de segurança em um sistema antes que eles possam ser explorados por atacantes**. Este processo é contínuo e essencial, especialmente em ambientes complexos como
- *serverless*, que interagem com múltiplos serviços.
- **Aprofundamento/Complemento:** As vulnerabilidades mais comuns em APIs de pagamento incluem:
    - **Injeção de SQL (SQL Injection):** Ocorre quando um atacante insere comandos SQL maliciosos em campos de entrada de dados. Se a aplicação não tratar esses dados corretamente, o comando pode ser executado no banco de dados, permitindo ao atacante visualizar, alterar ou deletar dados.
        - **Exemplo Prático:** Em um campo de busca de transação por ID, um atacante pode inserir `' OR '1'='1'`. Se a aplicação concatenar essa entrada diretamente em uma consulta SQL, o banco de dados pode retornar todas as transações em vez de apenas uma.
    - **Exposição de Dados Sensíveis:** Acontece quando informações críticas, como senhas, tokens de acesso ou números de cartão, são expostas em respostas de API, logs de sistema ou URLs sem a criptografia adequada.
        - **Exemplo Prático:** Uma API de consulta de perfil de usuário retorna todos os dados do cliente, incluindo o número do cartão de crédito completo. Ferramentas como o OWASP ZAP podem identificar esse tipo de vazamento durante um escaneamento.
    - **Controle de Acesso Deficiente (Broken Access Control):** Refere-se a falhas na validação de quem está acessando um recurso e quais permissões essa pessoa tem. Isso permite que um usuário comum acesse funcionalidades ou dados restritos a administradores.
        - **Exemplo Prático:** Um usuário A, ao solicitar os detalhes de seu pedido através do *endpoint* `/api/pedidos/123`, percebe que pode alterar o número no final da URL para `/api/pedidos/456` e visualizar os detalhes do pedido do usuário B, pois a API não verifica se o solicitante é o dono do pedido.

Semana 20 - Aula 2

Tópico Principal da Aula: Servidores web, API e segurança das aplicações

Subtítulo/Tema Específico: Arquitetura serverless

Código da aula: [SIS] ANO2C2B3S20A2

Objetivos da Aula:

- Implementar arquitetura *serverless* para otimizar a utilização de recursos.
- Compreender como serviços *serverless* simplificam a execução de código.
- Explorar os benefícios da arquitetura *serverless*, como escalabilidade e redução de custos.

Recursos Adicionais:

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.

Exposição do Conteúdo:

Referência do Slide: Slide 05 e 07 - Objetivos e Resumo da Aula

- **Definição:** **Arquitetura *serverless* (ou "sem servidor") é um modelo de desenvolvimento e execução de aplicações em nuvem no qual o provedor de nuvem (como AWS, Google Cloud ou Microsoft Azure) é responsável por gerenciar toda a infraestrutura de servidores. Os desenvolvedores se concentram apenas em escrever o código de suas funções, sem se preocupar com provisionamento, manutenção ou escalonamento de servidores.**
- **Aprofundamento/Complemento:** O termo "sem servidor" não significa que não existem servidores; significa apenas que os desenvolvedores não precisam gerenciá-los. O modelo mais comum de *serverless* é o **FaaS (Function as a Service)**, onde o código da aplicação é executado dentro de contêineres efêmeros (que são criados sob demanda e destruídos após o uso) acionados por eventos.
- **Exemplo Prático:** Uma aplicação de e-commerce precisa redimensionar as imagens dos produtos toda vez que um novo produto é cadastrado. Em vez de manter um servidor ligado 24/7 apenas para essa tarefa, pode-se usar uma função *serverless*. A função é acionada (inicia) somente quando uma nova imagem é salva em um serviço de armazenamento (o "evento"), executa o redimensionamento e depois desliga. O pagamento é feito apenas pelos milissegundos em que a função esteve em execução.

Referência do Slide: Slide 06 e 10 - Explorando arquiteturas serverless - AWS Lambda e Azure Functions

- **Definição:** **AWS Lambda e Azure Functions são as principais plataformas de FaaS do mercado. Elas permitem que os desenvolvedores executem código**

em resposta a eventos, como uma requisição HTTP a uma API, o upload de um arquivo ou uma mensagem em uma fila.

- **Aprofundamento/Complemento:**
    - **AWS Lambda:** É o serviço de computação *serverless* da Amazon Web Services. É amplamente utilizado devido à sua integração nativa com o vasto ecossistema de serviços da AWS (como S3 para armazenamento, API Gateway para APIs, e DynamoDB para banco de dados). É uma excelente escolha para quem já utiliza ou planeja utilizar a infraestrutura da AWS.
    - **Azure Functions:** É a solução *serverless* da Microsoft Azure. Oferece uma integração profunda com o ecossistema da Microsoft, incluindo serviços como Azure DevOps e Visual Studio. É frequentemente preferida por empresas que já têm uma forte dependência de tecnologias Microsoft.
- **Exemplo Prático:** Uma *fintech* quer notificar seus clientes via SMS sempre que uma transação acima de R$ 500 for realizada. Ela pode criar uma função no AWS Lambda que é acionada por um evento do banco de dados (nova transação registrada). A função verifica o valor da transação e, se for maior que R$ 500, utiliza outro serviço da AWS (o SNS) para enviar a mensagem de texto ao cliente.

Referência do Slide: Slide 07 - Benefícios e Uso da Arquitetura Serverless

- **Definição:** **A arquitetura *serverless* oferece vários benefícios que a tornam ideal para aplicações modernas, especialmente aquelas com cargas de trabalho variáveis.**
- **Aprofundamento/Complemento:**
    - **Escalabilidade Automática:** A plataforma de nuvem gerencia automaticamente o escalonamento da aplicação. Se mil usuários acessarem sua função simultaneamente, o provedor criará mil instâncias da função para atendê-los, sem qualquer intervenção manual.
    - **Redução de Custos (Modelo Pay-per-Use):** Você paga apenas pelo tempo de computação que realmente consome. Se sua função não for executada, não há custo. Isso elimina o desperdício de manter servidores ociosos.
    - **Foco no Código, Não na Infraestrutura:** Os desenvolvedores podem se concentrar em entregar valor de negócio através do código, em vez de gastar tempo com tarefas de gerenciamento de servidores, como atualizações de sistema operacional e patches de segurança.
- **Exemplo Prático:** Um site de venda de ingressos para shows tem picos de acesso enormes quando um novo evento é anunciado, mas fica quase inativo no resto do tempo. Com uma arquitetura tradicional, a empresa precisaria pagar por servidores potentes para suportar os picos, mesmo que eles ficassem ociosos 99% do tempo. Com *serverless*, a infraestrutura escala automaticamente durante os picos e reduz a zero quando não há acessos, gerando uma economia de custos significativa.
- **Vídeo Sugerido:** **Serverless // Dicionário do Programador**
- https://youtu.be/FaybjGx3uQI?si=IYx2qmViPZS6XjhA

Semana 20 - Aula 3

Tópico Principal da Aula: Servidores web, API e segurança das aplicações

Subtítulo/Tema Específico: Integração com serviços de pagamento

Código da aula: [SIS] ANO2C2B3S20A3

Objetivos da Aula:

- Integrar serviços de pagamento em aplicações back-end.
- Explorar práticas de segurança associadas à integração com APIs de pagamento.

- Aprender a gerenciar transações financeiras de forma eficiente.

Recursos Adicionais:

- Caderno para anotações;
- Computador com internet.

Exposição do Conteúdo:

Referência do Slide: Slide 06 - Integração com serviços de pagamento

- **Definição:** **A integração com serviços de pagamento envolve conectar uma aplicação (como um e-commerce ou sistema de agendamento) a uma API de um *gateway* de pagamento (como Stripe, Mercado Pago ou PagSeguro). Essa API permite que a aplicação processe transações financeiras (cartão de crédito, Pix, boleto) de forma segura e automatizada.**
- **Aprofundamento/Complemento:** O processo de integração geralmente segue estes passos:
  - **Escolha do Provedor:** Analisar taxas, documentação, recursos (ex: pagamento recorrente) e segurança.
  - **Obtenção de Credenciais:** Cadastrar-se no serviço para obter chaves de API (uma pública e uma secreta) que autenticam as requisições da sua aplicação.
  - **Implementação no Back-end:** Utilizar a documentação e os SDKs (Software Development Kits) do provedor para implementar a lógica de criação e gerenciamento de pagamentos.
  - **Tratamento de Respostas e Webhooks:** **Implementar rotinas para lidar com os diferentes status de uma transação (aprovada, recusada, pendente) e usar *webhooks* (notificações automáticas do servidor do gateway para o seu) para receber atualizações em tempo real.**
- **Exemplo Prático:** Uma loja virtual integra-se com a API do Mercado Pago. Quando um cliente finaliza a compra, o *front-end* da loja envia os dados do cartão de forma segura para o Mercado Pago, que retorna um *token*. O *back-end* da loja usa esse *token* e sua chave secreta para solicitar a criação do pagamento via API. O Mercado Pago processa a transação com o banco e informa ao *back-end* se o pagamento foi aprovado ou não.

Referência do Slide: Slide 07 a 12 - Práticas de Segurança e Vulnerabilidades (revisão)

- **Definição:** A segurança é o fator mais crítico na integração de pagamentos. Falhas podem resultar em perdas financeiras e danos à reputação da empresa. As práticas devem focar em proteger os dados em trânsito e em repouso.
- **Aprofundamento/Complemento:** Pontos cruciais de segurança:
    - **Segurança das Credenciais (API Keys):** As chaves secretas da API nunca devem ser expostas no código do *front-end* (lado do cliente). Elas devem ser armazenadas de forma segura no *back-end* e utilizadas apenas para comunicação servidor-a-servidor.
    - **Criptografia:** Toda a comunicação com a API de pagamento deve ocorrer sobre HTTPS (HTTP Seguro) para criptografar os dados em trânsito. Dados sensíveis, se precisarem ser armazenados (o que deve ser evitado), devem usar criptografia forte como o padrão AES-256.
    - **Conformidade PCI DSS:** O PCI DSS é um padrão de segurança da indústria de cartões de pagamento. A maioria dos *gateways* modernos cuida da maior parte da conformidade, processando e armazenando os dados do cartão em seus próprios ambientes seguros. A melhor prática é nunca deixar que dados brutos de cartão de crédito passem ou sejam armazenados em seus servidores.
    - **Validação de Entradas:** É essencial validar e higienizar todas as entradas de dados para prevenir ataques como **SQL Injection** e
    - **Cross-Site Scripting (XSS)**.
- **Exemplo Prático:** Em vez de coletar o número do cartão em um formulário no seu site e enviá-lo para o seu servidor, o *front-end* utiliza uma biblioteca do *gateway* de pagamento (ex: Stripe.js) que envia essa informação diretamente para os servidores do *gateway* de forma segura. O *gateway* retorna um *token* de uso único que representa o cartão. Sua aplicação usa esse *token*, que é seguro, para efetuar a cobrança, sem nunca ter tocado nos dados sensíveis do cartão.

Referência do Slide: Slide 06 e 13 - Gerenciamento de Transações

- **Definição:** Gerenciar o status das transações é fundamental para manter a integridade do sistema e fornecer feedback claro ao usuário. Cada transação passa por diferentes estados (ex:
- pendente, aprovada, recusada, estornada), e a aplicação precisa reagir corretamente a cada um deles.
- **Aprofundamento/Complemento:** Uma abordagem robusta para o gerenciamento de transações envolve registrar cada mudança de estado no banco de dados da aplicação. Por exemplo, quando um pagamento é criado, ele é salvo com o status pendente. Quando o *gateway* de pagamento confirma o sucesso da transação (geralmente via *webhook*), o status é atualizado para aprovado. Isso garante que, mesmo que o usuário feche o navegador, o sistema registrará o pagamento corretamente e liberará o produto ou serviço.
- **Exemplo Prático:** Um usuário paga um boleto gerado por sua aplicação. Inicialmente, a transação fica com o status pendente. Um ou dois dias depois, o banco confirma o pagamento ao *gateway*. O *gateway* envia uma notificação (*webhook*) para um *endpoint* específico da sua aplicação. Sua aplicação recebe

essa notificação, verifica sua autenticidade e atualiza o status do pedido no banco de dados para pago, disparando o processo de envio do produto e notificando o cliente por e-mail.
