Capítulo 5

Desenvolvimento ágil de software seguro e a cultura *DevSecOps*

Alexandre Braga[1] (CPQD)

Abstract

Software insecurity is a growing and recurring problem in society nowadays. All the time, vulnerabilities related to common defects and poor habits of insecure programming are exploited in attacks, affecting everyone's lives and leading to financial losses and various damages. As social activities, software security and development depend not only on technology but also on the engagement of everyone involved with it, since from the beginning of development to deployment and proper usage. This course shows how a combined approach of techniques, practices, and tools can enhance software security awareness, even during the early stages of agile development, incorporating a culture of prevention into the DevSecOps culture, where prevails the practices of automated testing, early detection of issues, and fast correction. To achieve this, a variety of complementary tools is necessary, distributed along a development pipeline, in a layered protection architecture, with scalable and repeatable results along multiple secure development cycles.

Resumo

A insegurança de software é um problema crescente e recorrente na sociedade hoje em dia. O tempo todo, vulnerabilidades relacionadas a defeitos comuns e maus hábitos de programação insegura são exploradas em ataques, afetando a vida de todos e levando a prejuízos financeiros e perdas diversas. Em sendo atividades sociais, segurança e desenvolvimento de software dependem não apenas da tecnologia, mas também do engajamento de todos os envolvidos no processo desde o início do desenvolvimento até a implantação e uso correto do software. Este minicurso mostra como a aplicação combinada de técnicas, práticas e ferramentas pode aumentar a conscientização sobre a segurança do software já durante os primeiros estágios de desenvolvimento ágil, incorporando a cultura de prevenção à cultura DevSecOps, em que imperam as práticas de testar automaticamente, detectar os problemas cedo e corrigi-los rápido.

[1]ambraga@cpqd.com.br

Para tal, é necessária uma variedade de ferramentas complementares, distribuídas ao longo de uma esteira de desenvolvimento, em uma arquitetura de proteção em camadas, com resultados escaláveis e repetíveis por vários ciclos de desenvolvimento seguro.

5.1. Introdução

À medida que aumenta a dependência das pessoas em relação à infraestrutura tecnológica, também aumentam a complexidade e a conectividade dos sistemas de computação subjacentes, porém, sem um aumento correspondente da segurança com que os softwares são construídos, configurados e implantados. Por isso, a insegurança dos sistemas de software é um problema crescente e recorrente. O tempo todo, vulnerabilidades relacionadas a defeitos comuns e maus hábitos de programação insegura são exploradas em ataques, afetando a vida de todos e levando a prejuízos financeiros e outras perdas.

Do ponto de vista do desenvolvedor de software, a segurança é um aspecto da qualidade e o desenvolvimento de software é uma atividade tanto técnica quanto social. Por outro lado, a garantia de segurança é uma responsabilidade coletiva. Logo, qualquer solução para a insegurança dos sistemas de software não é uma questão tecnológica apenas e deve considerar o engajamento das pessoas e a ajuda de todos os envolvidos no processo de construção desde o início do desenvolvimento até a implantação e uso correto dos sistemas.

Além disso, a escassez de mão de obra qualificada em segurança de software e a grande escala do problema dificultam o seu tratamento por times de segurança dedicados. Em um cenário comum, nas empresas de tecnologia, onde dezenas ou centenas de times de desenvolvimento atuam simultaneamente, uma equipe de segurança dedicada e reduzida tem dificuldades em atender a todas as demandas por mais segurança. A boa prática atual dita que esse time de segurança precisa dividir a responsabilidade e o protagonismo em segurança de software com os times de desenvolvimento, que devem ser capazes de resolver de forma autônoma os problemas simples, a fim de ganhar escala e velocidade no tratamento de vulnerabilidades técnicas, antes de acionar os especialistas.

Nesse contexto, existe um interesse crescente na academia e na indústria pelos aspectos práticos relacionados ao engajamento continuado de equipes de desenvolvedores de software em atividades de desenvolvimento de software seguro. Assim, este minicurso busca atender a demanda por conteúdo voltado para a utilização, diretamente por um esquadrão de desenvolvimento de software, de técnicas de análise de segurança apoiadas por ferramentas, tais como *Static Application Security Test* (SAST) e *Software Composition Analysis* (SCA), assim como também de testes de segurança em aplicações com auxílio de ferramentas *Dynamic Application Security Test* (DAST), em um contexto de *DevSecOps*, assim como também o uso de técnicas de segurança *by design* e de proteção da aplicação em produção, com o uso de *Web Application Firewalls* (WAFs).

O restante do texto está organizado da seguinte forma. A Seção 5.2 apresenta ao leitor a cultura de segurança de software ágil e aspectos de *DevSecOps*. A Seção 5.3 trata especificamente as avaliações e testes de segurança auxiliadas por ferramentas: revisão de código com SAST (subseção 5.3.2), avaliação de segurança com SCA (subseção 5.3.3) e testes de segurança com DAST (subseção 5.3.4). Já a Seção 5.4 trata vulnerabilidades de projeto com testes de segurança em APIs REST, enquanto a Seção 5.5 mostra como proteger aplicações web com WAF. Finalmente, a Seção 5.6 compara as ferramentas em relação aos falsos positivos e falsos

negativos e a Seção 5.7 tece considerações finais.

5.2. Segurança de software ágil e *DevSecOps*

Esta seção aborda as práticas de segurança de software adotadas pelas empresas de tecnologia que desenvolvem software conforme os métodos adaptativos [Institute 2023] e que incluem segurança em seus ciclos de desenvolvimento ágil. A seção discute como as práticas de segurança de software têm sido adotadas junto aos métodos ágeis e às esteiras ou pipelines *DevSecOps*.

A principal contribuição desta seção é a elaboração da ideia de esteira ou *pipeline DevSecOps* na qual as diversas técnicas e ferramentas de segurança são distribuídas de acordo com as necessidades de verificações manuais ou automatizadas, estáticas ou dinâmicas, antecipadas ou tardias, no ciclo de desenvolvimento de software.

Primeiramente, a seção discute os papéis principais envolvidos no desenvolvimento de software seguro, em particular, o *Security Champion*, e os aspectos de conscientização em segurança e cultura de desenvolvimento seguro. Em seguida, a seção discute o uso de ferramentas automáticas para acelerar e ganhar escala na execução de testes de segurança em esteiras de desenvolvimento *Continuous Integration*/*Continuous Delivery* (CI/CD) e a participação ativa de programadores na solução do débito de segurança. As técnicas e ferramentas são apresentadas e explicadas nesta seção.

5.2.1. O *Security Champion* de desenvolvimento de software seguro

No mundo atual repleto de incertezas e mudanças constantes, o desenvolvedor de software é um peregrino na sua jornada pessoal de aperfeiçoamento e aprendizado contínuo. De acordo com [Institute 2023], para sobreviver em um ambiente onde a incerteza de requisitos é grande e a velocidade da mudança tecnológica e cada vez maior, o desenvolvedor de software precisa seguir os princípios de desenvolvimento ágil, respondendo rápido às mudanças e sendo adaptativo no desenvolvimento de software para mitigar as incertezas.

Por outro lado, de acordo com [Pressman 2018], a flexibilidade de adaptação decorrente da assimilação rápida das mudanças leva naturalmente a uma maior instabilidade do software, com mais defeitos e um maior número de falhas. Somente o esforço deliberado do time de desenvolvimento pode trazer o software para próximo do estado estável anterior a mudança. Porém, uma vez que o software não é mais o mesmo porque a superfície de ataque mudou, o estado estável anterior é inatingível. Logo, há potencialmente mais defeitos que podem ser explotados como vulnerabilidades em ataques reais.

Segurança envolve pessoas realizando tarefas com eficiência por meio da tecnologia. O problema do estado atual de insegurança do software não é apenas uma questão tecnológica. As pessoas sempre fizeram coisas ruins, mesmo antes da haver tecnologias de computação e comunicação. A tecnologia facilitou o acesso aos meios para cometer crimes (cibernéticos) e compartilhar seus resultados. Quando a sociedade dependia menos da tecnologia, a necessidade por segurança cibernética também era menor. A partir do momento em que a sociedade ficou mais dependente da tecnologia, também aumentou a necessita de mais segurança cibernética e tanto o impacto quanto a percepção da insegurança do software aumentaram.

Segurança é importante para todo mundo! Aos usuário da tecnologia, as falhas de software trazem prejuízos pessoais e coletivos. Aos que constroem tecnologia, exige-se a respon-

sabilidade por mantê-la estável e segura para uso nos negócios e na sociedade. Quando os desenvolvedores de software não assumem a sua parte da responsabilidade pela segurança cibernética, aumenta a chance da tecnologia se tornar inadequada para exercer sua função na sociedade.

Toda pessoa desenvolvedora de software tem um papel a desempenhar na construção de software seguro, porque, no mínimo, a segurança é um aspecto de qualidade. Sendo assim, o desenvolvedor perfeiçoa-se em segurança de software, assume responsabilidade pela segurança do seu código, colabora com o grupo de segurança de aplicações da organização onde atua, colabora com o seu time na promoção da segurança de software e corrige vulnerabilidades simples diretamente.

O *Security Champion* [Miller 2022, saf 2019] vai além destas responsabilidades gerais, porque ele é a ponte entre os times de desenvolvimento e de segurança. Ele promove dentro do seu time a segurança *by design* desde cedo e facilita a comunicação entre desenvolvedores e especialistas em segurança de aplicações (e vice-versa). O *Security Champion* é um desenvolvedor que também sabe segurança. Como ele pertence ao time de desenvolvimento, ele é capaz de acelerar o andamento das tarefas, tratando as questões simples de segurança de software localmente e garantindo que verificações e testes são feitos direito e no tempo certo. Ele também leva problemas difíceis da equipe de software para os especialistas em segurança de aplicações.

Um *Security Champion* de desenvolvimento de software seguro não nasce pronto. Segurança cibernética é geralmente uma lacuna na formação das pessoas de tecnologia. Por isso, na prática, o desenvolvedor adquire habilidades e conhecimentos sobre segurança cibernética à medida em que supera os desafios de construir software seguro durante o desenvolvimento de software. O *Security Champion* não substitui o especialista em segurança de aplicações, mas deve conhecer o suficiente para fazer bem seu trabalho. Em uma espiral de aperfeiçoamento contínuo, no início, a vontade de aprender e de ajudar são mais importantes que conhecimento formal.

No geral, o *Security Champion* experiente precisa conhecer engenharia de software (métodos e ferramentas), modelagem de ameaças, tratamento de incidentes e vulnerabilidades, programação segura e testes de segurança. No papel de mediador entre desenvolvimento e segurança, muitas vezes, *soft skills* são importantes, tais como as habilidades em negociação, resolução de conflitos, apresentação, argumentação, motivação e engajamento de pessoas.

Em termos de cultura de segurança em times de software [Miller 2022, saf 2019], há papéis dentro do time de software mais alinhados às responsabilidades do *Security Champion*: arquitetos, engenheiros, desenvolvedores e testadores. Além disso, gerentes de projeto, agilistas, *product owners*, *scrum masters*, entre outros, também podem atuar como *Security Champions*, mas conflitos de interesse com outros papéis devem ser evitados.

Valem lembrar que, infelizmente, nem sempre o clima organizacional em um time de software é saudável. Em ambientes tóxicos, o *Security Champion* é menos eficaz em promover bons hábitos de segurança *by design*. Nesses ambientes, há três sinais de que o *Champion* não consegue mais ajudar sua equipe: *(i)* quando ele faz o trabalho do policial mau, aponta defeitos e vai embora, vigiando todo mundo e procurando culpados; *(ii)* quando ele é o bombeiro dos projetos, salva a segurança do projeto sozinho e age como o fiscal das regras de segurança; *(iii)* quando ele é o exército de uma pessoa só, tem inimigos declarados e luta uma batalha perdida. Por isso, é muito importante que o *Security Champion* tenha o apoio quase incondicional dos executivos, gestores e patrocinadores dos projetos, além do apoio do time.

5.2.2. Esquadrões, guildas, capítulos e tribos de segurança de software

Na cultura de software ágil, os desenvolvedores de software se organizam em diversas estruturas sociais. As organizações mais comuns são o Esquadrão, a Guilda, o Capítulo e a Tribo.

O time de desenvolvimento ágil é formado por todos os especialistas necessários e profissionais interdisciplinares, funcionando como um **esquadrão** autônomo e autocontido, com pouca dependência externa, capaz de resolver suas questões internamente de maneira rápida, direta e sem burocracia processual, que se recorrente, é logo automatizada. O esquadrão é geralmente uma equipe pequena (por exemplo, com até nove membros) multifuncional e auto-organizada, cujos membros possuem responsabilidades de ponta a ponta na esteira e trabalham juntos em direção a uma missão de longo prazo.

A **Tribo** pode ser entendida como um conjunto de esquadrões alinhados a um negócio, área de atuação, ou produto. Já o **Capítulo** lembra a organização de uma gerência e é um grupo formado por pessoas de competências comuns, como suporte de qualidade, mentoria ágil, segurança de aplicações, desenvolvimento web, etc. A **guilda**, por outro lado, é uma comunidade de interesse em que pessoas de toda a empresa se reúnem e compartilham conhecimento sobre uma área específica, como por exemplo, desenvolvimento de software seguro. Qualquer pessoa pode entrar ou sair de uma guilda a qualquer momento.

Os *Security Champions* geralmente se agrupam em uma guilda. Estabelecer uma função de *Security Champion* nos esquadrões empodera a segurança da aplicação enquanto o *Champion* mantém e aprimora suas habilidades e competências. Porém, o *Champion* precisa de uma comunidade de apoio. A guilda de *Security Champions* serve de plataforma para os *champions* desenvolverem habilidades com apoio do grupo e compartilharem conhecimento. Em sendo uma comunidade de vínculo fraco, a guilda de *Security Champions* ainda precisa de motivação e engajamento dos membros, além de alguma coordenação de ações. Geralmente, esta coordenação é realizada por um membro do grupo formal (Capítulo) de segurança de aplicações.

A guilda de *Security Champions* ajuda a segurança de software ágil quando dissemina o pensamento preventivo, o conhecimento sobre ameaças cibernéticas e a segurança desde o início do desenvolvimento, promovendo a segurança contínua e proativa versus segurança reativa e improvisada, estabelecendo melhores práticas e ferramentas comuns para todos os esquadrões, visando garantir que a segurança seja uma responsabilidade compartilhada e tratada por ada esquadrão. Uma cultura ágil forte influencia como o capítulo de segurança de aplicações está organizado e também como ele se relaciona com as outras estruturas sociais. Por exemplo, a guilda de *Security Champions* pode ser liderada e coordenada por um membro do capítulo de segurança de aplicações.

5.2.3. O grupo de segurança de aplicações

O capítulo de segurança de aplicações é um grupo organizado de especialistas em áreas da segurança de aplicações e tem sua própria organização interna. Hoje em dia, adota-se uma nomenclatura baseada em cores para identificar as especialidades e áreas de atuação deste grupo, que geralmente é composta pelos times vermelho, azul e amarelo. Opcionalmente, também há os times roxo, laranja e branco.

O time vermelho, ou *Red Team*, é o grupo de segurança ofensiva formado pelos *hackers* éticos. Este grupo possui visão adversarial e realiza ataques simulados furtivos (ocultos, sem

aviso) contra os controles de segurança, salvaguardas e contramedidas. As operações do *Red Team* servem para mostrar a insuficiência dos procedimentos de resposta à incidentes ou dos controles de segurança por meio dos testes não anunciados e furtivos (às escondidas). Antes de iniciar uma campanha de testes de intrusão, o *Red Team* deve definir a meta específica e as regras do jogo. Por exemplo, observando os seguintes pontos: ambiente de teste, janela de manutenção, não destruir a produção, etc. O resultado da operação do *Red Team* tem que ser útil para o *Blue Team* e para os desenvolvedores (*Yellow Team*). O *Blue Team* responde aos testes como se fossem incidentes reais.

O time azul, ou *Blue Team*, é o grupo de segurança defensiva. Grupo de defensores que tem o trabalho mais difícil: proteger os sistemas de software e os dados confidenciais tanto do *Red Team* quanto dos adversários reais. Atuando como *threat hunters*, procuram ativamente por ameaças e podem fazer as correções necessárias diretamente ou elaborar recomendações de melhoria, acompanhando a realização das correções. *Threat hunting* é a tarefa de identificação de ataques atuais ou passados e a contenção de ataques em andamento. Em resumo, o *Blue Team* se concentra na detecção de atividades maliciosas, proteção dos ativos de valor e manutenção da segurança. O *Blue Team* detecta, responde e se recupera dos incidentes de segurança, refinando e praticando a capacidade de responder aos incidentes. A capacidade de resposta à incidentes é obrigatória para o *Blue Team* funcionar como *Purple Team*.

O time roxo, ou *Purple Team*, é a combinação dos times vermelho e azul. O *Purple Team* combina ataques do *Red Team* e respostas do *Blue Team* em um esforço colaborativo e com ciclos de melhoria contínua. Os *Purple Teams* são úteis para equipes de segurança de aplicações com pouca capacidade de resposta à incidentes. Já os *Red Teams* são mais úteis para quem faz resposta aos incidentes de forma organizada (eficaz?). O *Purple Team* age como *Red Team* e *Blue Team* integrados e com comunicação próxima e interação fácil.

O *Purple Team* não é um *White Team*, que geralmente facilita a comunicação entre os times *Red* e *Blue* para supervisão e orientação. O *White Team* consiste de patrocinadores, facilitadores e outros *stakeholders*. O *White Team* não é técnico, não ataca e nem defende. Mas somente o *White Team* pode saber quando *Red Team* ataca.

Finalmente, o *Yellow Team*, ou time amarelo, é a denominação usada para se referir ao esquadrão de desenvolvimento ágil que vai implementar os controles de segurança e corrigir as vulnerabilidades; isto é, os desenvolvedores de software. A interface entre o *Yellow Team* e os times *Red* e *Blue* pode ser feita pelo time *Orange* (Laranja), que é responsável pelas ações de treinamento, capacitação e conscientização dos desenvolvedores sobre os métodos, práticas, ferramentas e tecnologias de segurança de software, além de ser o responsável pela guilda de *Security Champions*.

5.2.4. Desenvolvimento ágil de software seguro

Para Gary McGraw [McGraw 2004], segurança não é uma característica que pode ser simplesmente adicionada ao software. Em vez disso, ela seria uma propriedade emergente a partir de como o software é construído e usado. Assim, o software seguro é aquele que contempla com um grau alto de confiança justificada, mas não com certeza absoluta, um conjunto substancial de propriedades explícitas de segurança e de serviços de segurança [McGraw 2004]. Por isso, o software seguro resulta de um ciclo de desenvolvimento com segurança e de uma arquitetura de segurança, cujas propriedades emergentes se expressam durante o seu funcionamento.

Existem métodos, metodologias ou processos de desenvolvimento de software seguro, que são maneiras de organizar e conduzir as atividades de segurança durante o desenvolvimento de software, desde a concepção até a implantação da aplicação, incorporando tarefas, papeis e fluxos específicos para segurança. Nos últimos trinta anos, diversas destas metodologias foram propostas e experimentadas tanto pela academia quanto pela indústria. As mais conhecidas eram voltadas para os ambientes onde o desenvolvimento do software ocorre linearmente, em fases bem definidas e separadas no tempo, desde uma fase inicial de definição de objetivos e requisitos, passando pela elaboração de arquitetura, análise e projeto, implementação e testes, até a entrega e implantação em produção.

Um ponto chave de todas as metodologias é evitar que vulnerabilidades sejam incorporadas ao software, em vez de simplesmente descobrir e corrigir vulnerabilidades. Esta abordagem considera que o custo da descoberta tardia (em produção) de defeitos exploráveis como vulnerabilidades em ataques reais é muito alto. Por outro lado, o custo de correção de vulnerabilidades é menor quanto mais cedo ela ocorre no ciclo de vida da aplicação. Em se tratando de ataques cibernéticos, alguns custos são incalculáveis e irreparáveis, como por exemplo, a perda de vidas, a destruição de reputação e a exposição pública.

Uma vez que é mais barato descobrir as vulnerabilidades cedo, surge a ideia de *push left* ou *shift left*, que, ao olhar para um desenho de processo de software com atividades iniciais a esquerda e atividades finais a direita, adota a postura de antecipar as atividades de segurança (trazendo-as para o lado esquerdo do desenho sempre que possível). Exemplos de atividades de segurança que poderiam ser antecipadas são as seguintes: análise de segurança de arquitetura, especificação de requisitos de segurança, revisão de código (com ferramentas), avaliações de segurança e testes de intrusão.

Os métodos ágeis mudam a rigidez linear e sequencial do processo de desenvolvimento de software, fazendo com que em um período de tempo curto (por exemplo, uma *sprint* de 2 semanas), todas as fases sejam visitadas, por vezes de maneira concorrente e paralela, resultando em um software funcional (porém incompleto) a cada ciclo curto de desenvolvimento, evoluindo de maneira iterativa e incremental.

Uma vez que a cada ciclo curto há uma nova versão do software posta em execução, não é mais suficiente descobrir as vulnerabilidades cedo, mas também é importantíssimo corrigi-las rápido. No desenvolvimento ágil e seguro [Fisher 2022], a ideia *push left* evolui para o conceito *pushing left while accelerating right*, pregando que as atividades de segurança (todas elas) devem ser realizadas dentro de um ciclo curto (uma *sprint*). Isto significa que o rigor e complexidade das tarefas devem ser dosados a fim de torná-las mais rápidas, deixando refinamentos e detalhamentos para outros ciclos curtos. Por exemplo, em uma situação ideal, haveria um ciclo completo de verificações e testes de segurança, correções de vulnerabilidades e retestes a cada *sprint* de desenvolvimento [Fisher 2022].

Não há surpresa em perceber que o desenvolvimento ágil de software seguro somente é viável com grande uso de automação e o apoio dos *Security Champions*. Isto não significa que, por exemplo, testes de intrusão manuais, detalhados e demorados, realizados por *hackers* éticos não ocorram mais. Eles ainda acontecem, porém, fora do ciclo curto de desenvolvimento. Já as verificações e testes de segurança automatizados complementam os testes manuais e são usados para ganhar escala na testagem com detecção rápida dos problemas simples.

5.2.5. A esteira de desenvolvimento de software seguro e *DevSecOps*

DevSecOps [dev 2021] pode ser entendido como a integração contínua de princípios, processos e tecnologias de segurança (*Sec*) às práticas e processo da cultura *DevOps*, entendida como um movimento cultural formado por um conjunto de práticas e uma abordagem colaborativa que visa integrar o desenvolvimento de software (*Dev*) com as operações de infraestrutura e suporte (*Ops*). O objetivo principal do *DevOps* é aumentar a eficiência e a agilidade no desenvolvimento, entrega e operação de software, permitindo que as equipes de desenvolvimento e operações trabalhem de forma mais próxima e coordenada. A adoção plena da cultura *DevSecOps* é uma jornada com quatro momentos ou estágios de maturidade bem definidos. Em cada momento, a automação das atividades de segurança é adotada conforme a maturidade do time de desenvolvimento nas práticas *DevSecOps*.

No primeiro momento, o esquadrão ainda inexperiente adota um repositório de software e boas práticas de programação reforçadas na IDE e na geração de *build* (compilação e empacotamento), mas ainda não tem um pipeline ou esteira bem definida. A frequência de geração de *builds* pode ainda ser baixa e descontínua. Esta equipe deseja adotar as práticas de programação segura e faz isso com ferramentas SAST e SCA integrados à IDE, ao repositório de código fonte e possivelmente à geração da *build*.

No segundo momento, o esquadrão de software adota as práticas fundamentais de construção ou geração, integração e teste contínuos. Neste momento, já pode haver um *pipeline* automatizado que culmina na implantação do software em um ambiente de testes de aceitação ou homologação. Tudo que era feito por *scripts* isolados (incluindo as atividades de segurança) passa a ser realizado de maneira ordenada em um *pipeline*, que pode até ser interrompido se vulnerabilidades inaceitáveis forem detectadas. A existência de um ambiente de teste torna possível a realização de varreduras de vulnerabilidades automáticas com o auxílio de ferramentas DAST.

No terceiro momento, o esquadrão de software experiente realiza com desenvoltura e agilidade suas tarefas, atingindo o estágio de entrega contínua e implantação contínua. A automação das tarefas corriqueiras torna possível a sua realização com frequência alta. Neste momento, a construção e instalação automatizadas são apoiadas por processos automáticos de *failover* e *rollback*. Além disso, varreduras de vulnerabilidades e testes com ferramentas DAST são complementados, no ambiente de implantação (e produção) por teste *fuzzing* de injeção de falhas, testes de segurança interativos (*Interactive Application Security Tests* - IAST) e testes de intrusão (manuais).

Finalmente, no último estágio de maturidade *DevSecOps*, o esquadrão de software realiza todas as tarefas do estágio anterior com frequência alta, mais as ações de monitoramento contínuo e proteção em tempo de execução da aplicação no ambiente de produção. Neste momento, usam-se as ferramentas de autoproteção automática da aplicação (*Runtime Application Self-Protect* – RASP), como por exemplo, um WAF adaptativo integrado ao ambiente de execução da aplicação [da Silva et al. 2019] e capaz de fazer *patches* virtuais na aplicação em tempo real.

5.3. Avaliação e testes de segurança com auxílio de ferramentas

Esta seção está organizada como três tutoriais complementares de ferramentas usadas em avaliações e testes de segurança de software e que representam as categorias principais de ferramentas utilizadas em uma esteira *DevSecOps*: SAST, SCA e DAST. Ao aplicar ferramentas como parte

do processo de desenvolvimento, é possível fortalecer a segurança do software, identificando e corrigindo potenciais vulnerabilidades bem antes da implantação, garantindo assim um produto final mais confiável e protegido contra ameaças de segurança.

Avaliações e testes de segurança auxiliados por ferramentas aceleram a geração de resultados que podem ser tratados rapidamente. Porém, ferramentas automáticas não substituem o testador experiente. As ferramentas são softwares e podem ser incompletas e defeituosas, conforme estudos recentes [Braga et al. 2017, Braga et al. 2019]. Por isso, é necessário ter cautela na triagem de falsos positivos, evitando alarmes falsos que possam prejudicar a eficiência do processo. Além disso, as técnicas SAST, SCA e DAST são complementares entre si e, devem ser usadas em conjunto para mitigar falsos negativos (omissões), garantindo a identificação abrangente de vulnerabilidades.

A revisão de código com *Static Application Security Test* (SAST) é uma técnica essencial, apoiada por ferramentas, para a análise de vulnerabilidades no código-fonte, permitindo a detecção antecipada de possíveis falhas de segurança. Essa abordagem pode ser aplicada durante a fase de codificação, por meio da integração com IDEs, na compilação ou na geração de *builds*. Além disso, as ferramentas SAST podem ser integradas a pipelines CI/CD ou usadas de forma autônoma.

A avaliação de segurança com *Software Composition Analysis* (SCA) é uma técnica de verificação amplamente utilizada e apoiada por ferramentas para análise de vulnerabilidades em dependências de software, como componentes e bibliotecas de terceiros. Essa técnica é aplicada tanto na detecção antecipada de vulnerabilidades, durante as fases iniciais do desenvolvimento, quanto na detecção tardia, mais próxima à produção ou implantação do software. Por exemplo, é comum utilizar o SCA para realizar varreduras de vulnerabilidades em contêineres e imagens de implantação. Além disso, as ferramentas SCA também podem ser usadas para análise de licenciamento de software de código aberto (*Open Source Software* - OSS) e podem ser integradas a *pipelines* de CI/CD ou utilizadas de forma autônoma.

O teste de segurança com *Dynamic Application Security Test* (DAST) é uma técnica apoiada por ferramentas utilizada para analisar vulnerabilidades em tempo de execução de uma aplicação de software. Essa técnica é empregada para identificar vulnerabilidades próximas do momento de implantação ou entrega, ou seja, em fases tardias do ciclo de desenvolvimento. Para realizar a análise, é necessário que a aplicação em análise esteja em funcionamento em um ambiente de testes. A ferramenta DAST pode ser incorporada a *pipelines* CI/CD ou usada de forma autônoma como uma ferramenta independente.

5.3.1. Aplicações vulneráveis de aprendizado

Neste curso, as ferramentas SAST, SCA e DAST são exercitadas sobre aplicações propositalmente vulneráveis, livres e gratuitas, implementadas em uma variedade de tecnologias diferentes e utilizadas no aprendizado de técnicas de segurança, a saber: WebGoat em Java, bWAPP em PHP, Juice Shop com microsserviços em JavaScript e Gruyere em Python. Nem todas elas são utilizadas em todos os casos, algumas apenas nas aulas práticas em laboratório.

O OWASP Juice Shop é uma aplicação moderna construída em node.js com uma arquitetura de microsserviços. A página principal do projeto no OWASP está em `https://owasp.org/www-project-juice-shop/`. O código fonte está em `https://`

github.com/bkimminich/juice-shop. Um guia prático das tarefas de testes de segurança pode ser encontrado em https://pwning.owasp-juice.shop/. Há diversas opções de instalação ou implantação disponíveis em https://github.com/bkimminich/juice-shop#setup. Porém, a opção mais fácil e direta utiliza um contêiner Docker, com instruções em https://hub.docker.com/r/bkimminich/juice-shop.

A listagem 5.1 mostra os comandos para instalação por contêiner Docker. Quando a instalação é bem sucedida, a aplicação fica disponível em http://localhost:3000/.

Listagem 5.1. Juice Shop em Docker.

```
1 #Juice Shop em Docker
2 docker pull bkimminich/juice-shop
3 docker run -p 3000:3000 bkimminich/juice-shop
```

O OWASP WebGoat é a aplicação vulnerável tradicional do OWASP para o aprendizado de vulnerabilidades e foi desenvolvido em JavaEE. A página do WebGoat está em https://owasp.org/www-project-webgoat/. Este tutorial usa a versão 7.1 do WebGoat (para Java 8) que pode ser obtida em https://github.com/WebGoat/WebGoat. Um repositório com todo o código fonte do WebGoat 7.1, incluindo as lições e o programa principal, está em https://github.com/whoissqr/WebGoat-v71-all-in-one.

Também neste caso, a opção mais fácil e direta de instalação é por meio de um contêiner Docker. A listagem 5.2 mostra os comandos para instalação do WebGoat por contêiner Docker. Quando a instalação é bem sucedida, a aplicação fica disponível em http://localhost:8080/WebGoat.

Listagem 5.2. WebGoat em Docker.

```
1 #WebGoat em Docker
2 docker pull webgoat/webgoat-7.1
3 docker run -p 8080:8080 -t webgoat/webgoat-7.1
```

O bWAPP (*buggy Web APPlication*) é uma aplicação PHP sobre MySQL bastante utilizada em exercícios de análise de vulnerabilidades. O tutorial oficial do bWAPP está em https://www.mmebvba.com/sites/default/files/downloads/bWAPP_intro.pdf.

Outro tutorial interessante está em https://wooly6bear.files.wordpress.com/2016/01/bwapp-tutorial.pdf. Uma máquina virtual *standalone* do bWAPP está em https://sourceforge.net/projects/bwapp/files/bee-box/. Porém, a opção mais fácil e direta utiliza um contêiner Docker, com instruções em https://hub.docker.com/r/raesene/bwapp.

A listagem 5.3 mostra os comandos para instalação por contêiner Docker. Quando a instalação é bem sucedida, a aplicação fica disponível em http://localhost:8090/. Em seguida, deve-se ativar a aplicação pela url http://localhost:8090/install.php, seguir as instruções da página e entrar na aplicação com login "bee"e senha "bug".

Listagem 5.3. bWAPP em Docker.

```
1 docker pull raesene/bwapp
2 docker run -d -p 8090:80 raesene/bwapp
```

Gruyere é uma aplicação web vulnerável para aprendizado desenvolvida em Python que foi originalmente criada pelo Google e hoje é mantida por terceiros. O site oficial da

Gruyere (`https://google-gruyere.appspot.com/`) oferece um tutorial de testes de segurança apoiado por um laboratório, uma máquina virtual para implantação local da aplicação e o código fonte. A opção mais fácil é direta utiliza um contêiner Docker em `https://hub.docker.com/r/karthequian/gruyere`.

A listagem 5.4 mostra os comandos para instalação por contêiner Docker. Quando a instalação é bem sucedida, a aplicação fica disponível em `http://localhost:8008/`.

Listagem 5.4. Gruyere em Docker.

```
1 #Google Gruyere em Docker
2 docker pull karthequian/gruyere
3 docker run -d -p 8008:8008 karthequian/gruyere
```

5.3.2. Revisão de código auxiliada por ferramentas SAST

Esta seção trata revisão de código apoiado por ferramentas SAST livres e de código aberto como, por exemplo, Horusec (`https://horusec.io/`) e o Sonarqube (`https://www.sonarqube.org/`). A seção utiliza as ferramentas como comandos de linha e compara os resultados em nível macro; isto é, totalizações de vulnerabilidades gerais por criticidade (ou risco). Exemplos individualizados de vulnerabilidades são oferecidos apenas visando ilustração. A análise prática dos resultados é feita preferencialmente em sala de aula ou laboratório.

5.3.2.1. Análise estática de código-fonte com SonarQube *Community*

O SonarQube *community* (`https://www.sonarqube.org/`) é uma ferramenta para inspeção contínua da qualidade do código, incluindo análise estática de código para uma grande variedade de linguagens, detecção de defeitos e vulnerabilidades e métricas de qualidade. Já o SonarLint (`https://www.sonarlint.org`) é uma extensão para IDE de código aberto que realiza inspeção de código durante a codificação. Como um corretor ortográfico, o SonarLint marca falhas e fornece *feedback* em tempo real e orientações de correção. É possível sincronizar o SonarLint com o perfil definido no SonarQube. A versão *community* tem mostrado bons resultados para detecção de violação de boas práticas de codificação, duplicação de código, complexidade excessiva, exibir métricas de código e cobertura.

No entanto, para a identificação de vulnerabilidades, há ressalvas. A edição *community* não possui o mesmo conjunto de regras das edições comerciais (*Developer*, *Cloud* e *Enterprise*). Por isso, observam-se mais falsos positivos (alarmes falsos) e falsos negativos (omissões) na edição comunitária que em outras edições, além de questões de integração de *scanners* modernos e atualização das regras para tecnologias de software mais novas. Esta situação diminui a confiança e prejudica a adoção da ferramenta como SAST. Por outro lado, existe uma comunidade de usuários bastante ativa e que pode ser consultada na busca por soluções para as lacunas da edição gratuita em situações específicas.

A distribuição gratuita para a comunidade de desenvolvedores pode ser obtida em `https://www.sonarsource.com/open-source-editions/`. Há três opções de instalação complementares: o SonarLint, um *plug-in* para integração em IDEs; o *SonarQube Server*, uma aplicação administrativa de gestão de varreduras e o `sonar-scanner`, o *scanner* de vulnerabilidade propriamente dito. O aluno interessado deve procurar a documentação de cada opção. As listagens a seguir mostram a utilização do `sonar-scanner`.

A listagem 5.5 mostra o comando para realização de uma varredura de vulnerabilidades com a ferramenta SAST SonarQube *Community* sobre o código-fonte da aplicação Juice Shop. Primeiramente o comando `pwd` mostra que o *prompt* de comando está na pasta `sast/-sources`. Em seguida, o comando `ls -l` lista o conteúdo da pasta `/sast/sources`, revelando uma pasta de código fonte para cada uma das aplicações vulneráveis. Já o comando `cd juice-shop-master/` posiciona o *prompt* na pasta do Juice Shop, enquanto o comando `sonar-scanner` realiza a varredura de vulnerabilidades com os seguintes parâmetros específicos:

- `-Dsonar.projectKey=juice-shop` é o identificador do projeto na aplicação administrativa do sonar e é criado junto com o projeto de análise de código fonte na ferramenta.
- `-Dsonar.sources=.` é a pasta corrente onde está o código-fonte.
- `-Dsonar.host.url=http://localhost:9000` é a URL da aplicação administrativa do sonar que vai receber o resultado da avaliação de segurança.
- `-Dsonar.login=sqa_d0d5432 ... 7a4f2b73b` é o *token* de autenticação usado pelo *scanner* no acesso autorizado à *console* administrativa. Este *token* é obrigatório e pode ser gerado durante a criação do projeto de *scan* ou apenas uma vez.

Se a instalação do SonarQube *out-of-the-box* NÃO mostra alertas para o Juice Shop, isso pode ser um defeito de configuração do *scanner*, que não é capaz de detectar problemas de segurança em *Typescrypt* apenas com a configuração *default*. Sabe-se que as vulnerabilidades existem, então as ausências são omissões (falsos negativos) do SonarQube. Por outro lado, o SonarQube mostra *Security Hotspots* para o Juice Shop nas seguintes categorias OWASP Top 10 2017: A1 (*Injection*), A2 (*Broken Authentication*), A3 (*Sensitive Data Exposure*), A6 (*Security Misconfiguration*). O aluno interessando deve procurar na comunidade pelas configurações de regras de segurança para *Typescrypt*.

Listagem 5.5. Varredura de código-fonte do Juice Shop com o SonarQube.

```
1 # scan do juiceshop
2 $ pwd
3 ./sast/sources
4 $ ls -l
5 total 20
6 drwxr-xr-x  6 ******* domain users 4096 mar 13 15:20 bwapp-code-master
7 drwxr-xr-x  4 ******* domain users 4096 mar 13 14:59 gruyere-app
8 drwxr-xr-x 24 ******* domain users 4096 abr 26 10:02 juice-shop-master
9 drwxr-xr-x  7 ******* domain users 4096 mar 13 16:02 secDevLabs-master
10 drwxr-xr-x 73 ******* domain users 4096 jul 24 14:54 WebGoat-v71-all-in-one-master
11 $ cd juice-shop-master/
12 $ sonar-scanner \
13   -Dsonar.projectKey=juice-shop \
14   -Dsonar.sources=. \
15   -Dsonar.host.url=http://localhost:9000 \
16   -Dsonar.login=sqa_d0d5432 ... e1c7a4f2b73b
```

A listagem 5.6 mostra o comando para uma varredura de vulnerabilidades com a ferramenta SAST SonarQube *Community* sobre o código-fonte da aplicação WebGoat 7.1. Primeiramente o comando `pwd` mostra que o *prompt* de comando está na pasta `/sast/sources`. Em seguida, comando `cd WebGoat-v71-all-in-one-master` posiciona o *prompt*

na pasta do WebGoat, enquanto o comando `sonar-scanner`, com parâmetros específicos, realiza a varredura de vulnerabilidades. O comando `sonar-scanner` utiliza os seguintes parâmetros específicos (diferentes da execução anterior):

- `-Dsonar.projectKey=WebGoat` é o identificador do projeto de *scan* na aplicação administrativa do sonar.
- `-Dsonar.sources=.` é a pasta corrente onde está o código-fonte.

No geral, o SonarQube mostra muitos alertas de vulnerabilidades do OWASP Top 10 de 2017 para o WebGoat. Entre eles, estão os seguintes: A3 *Sensitive Data Exposure*, A1 *Injection*, A6 *Security Misconfiguration*, A4 *XML External Entities* (XXE) e A5 *Broken Access Control*. O SonarQube também mostra muitos *Security Hotspots* do OWASP Top 10 de 2021 para o WebGoat, a saber: A1 (*Broken Access Control*), A2 (*Cryptographic Failures*), A3 (*Injection*), A4 (*Insecure Design*), A5 (*Security Misconfiguration*), A7 (*Identification and Authentication Failures*). O resultado da análise depende da versão do *scanner*, do acesso ao código fonte e da configuração da aplicação. Além disso, muitos alertas podem ser falsos positivos.

Listagem 5.6. Varredura de código-fonte do WebGoat 7.1 com o SonarQube.

```
1 # scan do webgoat
2 $ pwd
3 ./sast/sources
4 $ cd WebGoat-v71-all-in-one-master
5 $ sonar-scanner \
6   -Dsonar.projectKey=WebGoat \
7   -Dsonar.sources=. \
8   -Dsonar.java.binaries=../sast/binaries/webgoat-container-7.1-exec.jar\
9   -Dsonar.host.url=http://localhost:9000 \
10  -Dsonar.login=sqa_d0d5432 ... 7a4f2b73b
```

A listagem 5.7 mostra o comando para realização de uma varredura de vulnerabilidades com a ferramenta SAST SonarQube *Community* sobre o código-fonte da aplicação bWAPP. Primeiramente o comando `pwd` mostra que o *prompt* de comando está na pasta `/sast/-sources`. Em seguida, o comando `cd bwapp-code-master` posiciona o *prompt* na pasta do bWAPP, enquanto o comando `sonar-scanner` realiza a análise com parâmetros específicos, a saber:

- `-Dsonar.projectKey=bWAPP` é o identificador do projeto de *scan* na aplicação administrativa do sonar.
- `-Dsonar.sources=.` é a pasta corrente onde está o código-fonte.

O SonarQube na configuração padrão pode não mostrar alertas para o bWAPP, mas associa vários *Security Hotspots* ao OWASP Top 10 2017, a saber: A1 (*Injection*), A2 (*Broken Authentication*), A3 (*Sensitive Data Exposure*) (criptografia fraca), A6 (*Security Misconfiguration*), A7 *Cross-Site Scripting (XSS)*), A9 (*Components with vulnerabilities*) e A10 (*Insufficient Logging and Monitoring*). O aluno interessando deve procurar na comunidade SonarQube pelas configurações de regras de segurança para PHP.

Listagem 5.7. Varredura de código-fonte do bWAPP com o SonarQube.

```
1 # scan do bWAPP
2 $ pwd
3 ./sast/sources
4 $ cd bwapp-code-master
5 sonar-scanner \
6   -Dsonar.projectKey=bWAPP \
7   -Dsonar.sources=. \
8   -Dsonar.host.url=http://localhost:9000 \
9   -Dsonar.login=sqa_d0d5432 ... 7a4f2b73b
```

A listagem 5.8 mostra o comando para realização de uma varredura de vulnerabilidades com a ferramenta SAST SonarQube *Community* sobre o código-fonte da aplicação Gruyere. Com o *prompt* na pasta `sast/sources/gruyere-app`, o comando `sonar-scanner` é executado com parâmetros específicos:

- `-Dsonar.projectKey=Gruyere` é o identificador do projeto de *scan* na aplicação administrativa do sonar.
- `-Dsonar.sources=.` é a pasta corrente onde está o código-fonte.

O *scanner* padrão do SonarQube pode não mostrar alertas nem *Security Hotspots* para o *Gruyere*. O resultado da análise depende da versão do *scanner*, do acesso ao código fonte e da configuração da aplicação. Há nitidamente uma deficiência do *scanner* padrão para a linguagem Phyton, resultando em falsos negativos (omissões). Assim como das vezes anteriores, para obter resultados mais detalhados, aluno interessando deve procurar na comunidade pelas configurações de regras de segurança específicas de Python.

Listagem 5.8. Varredura de código-fonte do Gruyere com o SonarQube.

```
1 #scan do Gruyere
2 $ pwd
3 ./sast/sources
4 $ cd gruyere-app
5 $ sonar-scanner \
6   -Dsonar.projectKey=Gruyere \
7   -Dsonar.sources=. \
8   -Dsonar.host.url=http://localhost:9000 \
9   -Dsonar.login=sqa_d0d5432 ... 7a4f2b73b
```

5.3.2.2. Análise estática de código-fonte com HoruSec

O Horusec (`https://horusec.io/`) é uma ferramenta de análise estática de código aberto, projetada para identificar vulnerabilidades de segurança durante a fase de desenvolvimento. Com suporte para as principais linguagens de programação, o Horusec auxilia os desenvolvedores na detecção e correção proativa de vulnerabilidades em seu código. Além disso, o Horusec funciona como um integrador de vários *scanners* de vulnerabilidades, permitindo a consolidação de resultados e simplificando o processo de análise de segurança.

Uma vez instalado, a utilização do HoruSec *standalone* por linha de comando é bastante simples. As listagens 5.9, 5.10, 5.11 e 5.12 mostram trechos dos resultados das análises do HoruSec sobre os códigos-fonte das aplicações vulneráveis. Em todos os casos, a ferramenta é ativada a partir da pasta de código fonte da aplicação com o comando `horusec start -p . --disable-docker="true"`, com a opção Docker desligada.

A varredura com mais vulnerabilidades de risco alto (críticas e altas) foi a do Juice Shop (364 + 101 = 465), seguida pela do WebGoat em segundo (62 + 193 = 255) e pela do bWAPP em terceiro (147 + 11 = 158). A Gruyere ficou em último com apenas uma vulnerabilidade crítica descoberta. No geral, muitos alertas podem ser falsos positivos, mas todos os exemplos de vulnerabilidades mostrados nas listagens são positivos verdadeiros. Uma vez que sabemos que Gruyere possui várias vulnerabilidades exploráveis, pode-se concluir que, neste caso, a varredura com o HorusSec possui vários falsos negativos (omissões).

Listagem 5.9. Varredura de código-fonte do Juice Shop com o HoruSec.

```
# scan do juiceshop
$ pwd
./sast/sources
$ cd juice-shop-master/
$ horusec start -p . --disable-docker="true"

. . .
# Exemplo de vulnerabilidade encontrada
================================================================================

Language: Leaks
Severity: CRITICAL
Line: 80
Column: 41
SecurityTool: HorusecEngine
Confidence: MEDIUM
File: ./frontend/src/app/Services/two-factor-auth-service.spec.ts
Code: expect(req.request.body).toEqual({ password: 's3cr3t!' })
RuleID: HS-LEAKS-26
Type: Vulnerability
ReferenceHash: edeff4a1dc1234ed7ea5cb2c5e8b6c7065f802776470fe12985c209296162382
Details: (1/1) * Possible vulnerability detected: Hard-coded password
The software contains hard-coded credentials, such as a password or cryptographic
key, which it uses for its own inbound authentication, outbound communication to
external components, or encryption of internal data. For more information checkout
the CWE-798 (https://cwe.mitre.org/data/definitions/798.html) advisory.

================================================================================

In this analysis, a total of 474 possible vulnerabilities were found and we
classified them into:
Total of Vulnerability CRITICAL is: 364
Total of Vulnerability HIGH is: 101
Total of Vulnerability MEDIUM is: 5
Total of Vulnerability LOW is: 4
```

Listagem 5.10. Varredura de código-fonte do WebGoat 7.1 com o HoruSec.

```
# scan do webgoat
$ pwd
./sast/sources
$ cd WebGoat-v71-all-in-one-master
$ horusec start -p . --disable-docker="true"

. . .
# Exemplo de vulnerabilidade encontrada
================================================================================

Language: JavaScript
Severity: CRITICAL
Line: 22
Column: 25
SecurityTool: HorusecEngine
Confidence: MEDIUM
File: ./xxe/src/main/resources/plugin/XXE/js/xxe.js
Code: var result = eval('(' + req.responseText + ')');
```

```
RuleID: HS-JAVASCRIPT-2
Type: Vulnerability
ReferenceHash: 2dc6ab35cd1e763bb67697b71c824079e8672604b83e78ccd74cef3938eae043
Details: (1/1) * Possible vulnerability detected: No use eval
The eval function is extremely dangerous. Because if any user input is not handled
correctly and passed to it, it will be possible to execute code remotely in the
context of your application (RCE - Remote Code Executuion). For more information
checkout the CWE-94 (https://cwe.mitre.org/data/definitions/94.html) advisory.

================================================================================

In this analysis
     , a total of 299 possible vulnerabilities were found and we classified them into:
Total of Vulnerability LOW is: 5
Total of Vulnerability CRITICAL is: 62
Total of Vulnerability HIGH is: 193
Total of Vulnerability MEDIUM is: 39
```

Listagem 5.11. Varredura de código-fonte do bWAPP com o HoruSec.

```
# scan do bWAPP
$ pwd
./sast/sources
$ cd bwapp-code-master
$ horusec start -p . --disable-docker="true"

...
# Exemplo de vulnerabilidade encontrada
================================================================================

Language: JavaScript
Severity: CRITICAL
Line: 465
Column: 20
SecurityTool: HorusecEngine
Confidence: MEDIUM
File: ./bWAPP/js/json2.js
Code: j = eval('(' + text + ')');
RuleID: HS-JAVASCRIPT-2
Type: Vulnerability
ReferenceHash: 928b1f21a15339da9b0296fe25efa13a8908b97667280a0d6f4bed00b24ad1bf
Details: (1/1) * Possible vulnerability detected: No use eval
The eval function is extremely dangerous. Because if any user input is not handled
correctly and passed to it, it will be possible to execute code remotely in the
context of your application (RCE - Remote Code Executuion). For more information
checkout the CWE-94 (https://cwe.mitre.org/data/definitions/94.html) advisory.

================================================================================

In this analysis, a total of 158 possible vulnerabilities were found and we
classified them into:
Total of Vulnerability CRITICAL is: 147
Total of Vulnerability HIGH is: 11
```

Listagem 5.12. Varredura de código-fonte do Gruyere com o HoruSec.

```
#scan do Gruyere
$ pwd
./sast/sources
$ cd gruyere-app
$ horusec start -p . --disable-docker="true"
. . .

# Exemplo de vulnerabilidade encontrada
================================================================================

Language: JavaScript
Severity: CRITICAL
Line: 25
```

```
13 Column: 6
14 SecurityTool: HorusecEngine
15 Confidence: MEDIUM
16 File: ./resources/lib.js
17 Code: eval('(' + httpRequest.responseText + ')');
18 RuleID: HS-JAVASCRIPT-2
19 Type: Vulnerability
20 ReferenceHash: 985b9e6639f077e0c29136190bbc01be3a187bf6da24f635a90efa0f73d018a1
21 Details: (1/1) * Possible vulnerability detected: No use eval
22 The eval function is extremely dangerous. Because if any user input is not handled
23 correctly and passed to it, it will be possible to execute code remotely in the
24 context of your application (RCE - Remote Code Execution). For more information
25 checkout the CWE-94 (https://cwe.mitre.org/data/definitions/94.html) advisory.
26
27 ================================================================================
28
29 In this analysis, a total of 1 possible vulnerabilities were found and we
30 classified them into:
31 Total of Vulnerability CRITICAL is: 1
```

5.3.3. Avaliação de segurança auxiliada por ferramentas SCA

Esta seção trata ferramentas SCA livres e gratuitas, tais como OWASP Dependency-Check (`https://owasp.org/www-project-dependency-check`) e o Trivy (`https://trivy.dev/`) (especialmente quando aplicado a imagens Docker). A seção utiliza as ferramentas como comandos de linha e compara os resultados em nível macro; isto é, totalizações de vulnerabilidades gerais por criticidade. Exemplos individualizados de vulnerabilidades são oferecidos apenas visando ilustração. A análise prática dos resultados é feita preferencialmente em sala de aula ou laboratório.

5.3.3.1. Avaliação de segurança com o OWASP Dependency-Check

O Dependency-Check (`https://owasp.org/www-project-dependency-check`) é uma ferramenta livre e de código aberto do OWASP e faz a varredura de vulnerabilidades antecipada de aplicações para encontrar componentes e dependências vulneráveis nas fases iniciais do desenvolvimento. Ela atua sobre código binário da aplicação é complementar a outras ferramentas similares, como o Trivy, que é indicado para etapas tardias do pipeline. Trata o item A06 (*Vulnerable and Outdated Components*) da lista Top 10 2021 do OWASP e suporta várias linguagens de programação, podendo ser integrado a *pipelines* de CI/CD.

A avaliação de segurança com o Dependency-Check é feita com o comando de linha `dependency-check.sh --scan <<pasta com bibliotecas>>`, que produz como saída um relatório em `html`. As avaliações das quatro aplicações vulneráveis pelo Dependency-Check apresentaram resultados esclarecedores. A análise foi bem sucedida apenas em relação ao Juice Shop e ao WebGoat, são aplicações web com arquitetura moderna e fortemente baseada em componentes. Isto faz sentido, uma vez que a preocupação com dependências inseguras é relativamente nova e relacionada ao uso crescente de componentes de terceiros no software atual.

A Tabela 5.1 mostra as vulnerabilidades por dependência, indicando também a quantidade de CVEs associados à dependência, o grau de confiança na análise e a quantidade de evidências encontradas (que justificam a confiança). No Juice Shop, foram encontradas 5 vulnerabilidade críticas e 15 altas, de um total de 30 vulnerabilidades, em componentes variados, todas

Tabela 5.1. Vulnerabilidades encontradas com o Dependency-Check.

Aplicação	Dependência	Severidade	CVEs	Confiança	Evidências
Juice Shop	bench.js	HIGH	1		3
Juice Shop	blaze.jar	MEDIUM	1	Highest	86
Juice Shop	dottie:2.0.3	HIGH	1	Highest	9
Juice Shop	express-jwt:0.1.3	CRITICAL	1	Highest	10
Juice Shop	hbs:4.2.0	MEDIUM	1	Highest	10
Juice Shop	jquery.js	MEDIUM	6		3
Juice Shop	jsonwebtoken:0.4.0	CRITICAL	4	Highest	8
Juice Shop	moment.js	HIGH	4		3
Juice Shop	moment.min.js	HIGH	4		3
Juice Shop	notevil:1.3.3	MEDIUM	1	Highest	8
Juice Shop	request:2.88.2	MEDIUM	1	Highest	9
Juice Shop	sanitize-html:1.4.2	HIGH	4	Highest	8
Juice Shop	semver:7.3.8	HIGH	1		8
WebGoat	bootstrap.min.js	MEDIUM	7		3
WebGoat	jquery-1.10.2.min.js	MEDIUM	6		3
WebGoat	jquery-ui-1.10.4.custom.min.js	MEDIUM	5		5
WebGoat	underscore-min.js	HIGH	1		3
WebGoat	underscore.js	HIGH	1		3

elas com CVEs identificados. Já no WebGoat, duas (2) vulnerabilidades altas foram encontradas, de um total de 20 vulnerabilidades. Não houve relato de vulnerabilidades nas outras aplicações.

Vale observar que nestes casos, em se tratando de componentes externos de terceiros e fora do controle do time de desenvolvimento das aplicações analisadas, a estratégia de correção recomendada é a atualização das dependências vulneráveis por versões mais seguras, observando-se a manutenção do código fonte que utiliza a dependência atualizada. Além disso, a ferramenta Dependency-Track (`https://dependencytrack.org/`) pode ser usada quando o objetivo final é a gestão de segurança das dependências, no conceito de *Software Bill of Materials* (SBOM), indo além da análise de segurança de componentes de SCA.

5.3.3.2. Avaliação de segurança com o Trivy

O Trivy (`https://trivy.dev/`) é uma ferramenta gratuita para fazer a varredura de vulnerabilidades em imagens Docker, sistema de arquivos e arquivos de Infraestrutura como Serviço (*infrastructure as a Service* – IaC) e geração e verificação de *Software Bill of Materials* (SBOM). Ela pode ser adicionada a esteira para verificação tardia de vulnerabilidades em imagens Docker e rodar localmente durante o desenvolvimento das imagens de contêiner. A ferramenta deve ser usada em conjunto com outras para aumentar a cobertura da segurança das aplicações.

Na listagem 5.2, o comando `docker images` lista as imagens instaladas das quatro aplicações vulneráveis. As saídas dos comandos foram editados para melhorar a formatação e o uso do espaço. A varredura de segurança com Trivy sobre imagens Docker é feita com a linha de comando `trivy image NOME-IMAGEM-DOCKER`. A listagem mostra os comandos Trivy para cada uma das imagens, nas linhas 8, 19, 38 e 49. Em cada caso, o resultado da varredura é direcionada para um arquivo correspondente à aplicação analisada. Todas as varreduras fazem análises de vulnerabilidades e de segredos dos componentes da aplicação e também do

Tabela 5.2. Vulnerabilidades críticas e altas encontradas com o Trivy.

Aplicação	Componentes	Total geral	Total Críticas	Total Altas
Juice Shop	bkimminich/juice-shop	21	0	4
Juice Shop	Node.js (node-pkg)	66	7	14
Juice Shop	/juice-shop/lib/insecurity.ts (secrets)	1	0	1
Juice Shop	**Totais**	88	7	19
WebGoat	webgoat/webgoat-7.1 (debian 8.6)	513	76	169
WebGoat	Java (jar)	118	36	44
WebGoat	**Totais**	631	112	213
bWAPP	raesene/bwapp (ubuntu 14.04)	1778	0	28
bWAPP	/private/ssl-cert-snakeoil.key (secrets)	1	0	1
bWAPP	**Totais**	1779	0	29
Gruyere	karthequian/gruyere (alpine 3.3.3)	1	1	0
Gruyere	**Totais**	1	1	0

sistema operacional da imagem.

A tabela 5.2 resume as vulnerabilidades identificadas pelo Trivy nestas avaliações. No Juice Shop, 88 CVEs foram identificadas na imagem e no node.js, sendo 7 críticas e 19 altas. No WebGoat, foram 631 CVEs na imagem e arquivos Jar, sendo 112 críticas e 213 altas. No bWAPP, foram 1779 CVEs na imagem, com apenas 28 altas. Gruyere apresentou o resultado modesto, com apenas uma CVE crítica. De modo geral, Trivy gerou mais alertas de vulnerabilidades que Dependency-Check.

Listagem 5.13. Varredura de imagens Docker com Trivy.

```
$ docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
bkimminich/juice-shop         latest    8493311c32c9   5 months ago    580MB
webgoat/webgoat-7.1           latest    20fef7540300   6 years ago     460MB
raesene/bwapp                 latest    8be28fba48ec   7 years ago     441MB
karthequian/gruyere           latest    4ed76f33b77f   3 years ago     210MB
$
$ sudo trivy image bkimminich/juice-shop > ./trivy-juice.txt
11:56:05  INFO  Vulnerability scanning is enabled
11:56:05  INFO  Secret scanning is enabled
11:56:05  INFO
       If your scanning is slow, please try '--scanners vuln' to disable secret scanning
11:56:05  INFO  Please see also https://aquasecurity.github
     .io/trivy/v0.43/docs/scanner/secret/#recommendation for faster secret detection
11:56:07  INFO  Detected OS: debian
11:56:07  INFO  Detecting Debian vulnerabilities...
11:56:07  INFO  Number of language-specific files: 1
11:56:07  INFO  Detecting node-pkg vulnerabilities...
11:56:07  INFO  Table result includes only package
      filenames. Use '--format json' option to get the full path to the package file.
$
$ sudo trivy image webgoat/webgoat-7.1    > ./trivy-webgoat.txt
11:57:190 INFO  Vulnerability scanning is enabled
11:57:19  INFO  Secret scanning is enabled
11:57:19  INFO
       If your scanning is slow, please try '--scanners vuln' to disable secret scanning
11:57:19  INFO  Please see also https://aquasecurity.github
     .io/trivy/v0.43/docs/scanner/secret/#recommendation for faster secret detection
11:57:34  INFO  JAR files found
11:57:34  INFO  Java DB Repository: ghcr.io/aquasecurity/trivy-java-db:1
11:57:34  INFO  Downloading the Java DB...
446.69 MiB / 446.69 MiB
      [-------------------------------------------------] 100.00% 7.27 MiB p/s 1m2s
11:58:37  INFO  The Java DB is cached for 3 days. If you
     want to update the database more frequently, the '--reset' flag clears the DB cache.
```

```
29 11:58:37  INFO  Analyzing JAR files takes a while...
30 11:58:38  INFO  Detected OS: debian
31 11:58:38  INFO  Detecting Debian vulnerabilities...
32 11:58:38  INFO  Number of language-specific files: 1
33 11:58:38  INFO  Detecting jar vulnerabilities...
34 11:58:38  WARN  This OS version is no longer supported by the distribution: debian 8.6
35 11:58:38  WARN  The vulnerability
      detection may be insufficient because security updates are not provided
36 11:58:38  INFO  Table result includes only package
      filenames. Use '--format json' option to get the full path to the package file.
37 $
38 $ sudo trivy image raesene/bwapp         > ./trivy-bwapp.txt
39 12:01:50  INFO  Vulnerability scanning is enabled
40 12:01:50  INFO  Secret scanning is enabled
41 12:01:50  INFO
       If your scanning is slow, please try '--scanners vuln' to disable secret scanning
42 12:01:50  INFO  Please see also https://aquasecurity.github
     .io/trivy/v0.43/docs/scanner/secret/#recommendation for faster secret detection
43 12:02:15  INFO  Detected OS: ubuntu
44 12:02:15  INFO  Detecting Ubuntu vulnerabilities...
45 12:02:15  INFO  Number of language-specific files: 0
46 12:02:15  WARN  This OS version is no longer supported by the distribution: ubuntu 14.04
47 12:02:15  WARN  The vulnerability
      detection may be insufficient because security updates are not provided
48 $
49 $ sudo trivy image karthequian/gruyere   > ./trivy-gruyere.txt
50 12:02:49  INFO  Vulnerability scanning is enabled
51 12:02:49  INFO  Secret scanning is enabled
52 12:02:49  INFO
       If your scanning is slow, please try '--scanners vuln' to disable secret scanning
53 12:02:49  INFO  Please see also https://aquasecurity.github
     .io/trivy/v0.43/docs/scanner/secret/#recommendation for faster secret detection
54 12:02:56  INFO  Detected OS: alpine
55 12:02:56  INFO  Detecting Alpine vulnerabilities...
56 12:02:56  INFO  Number of language-specific files: 1
57 12:02:56  INFO  Detecting python-pkg vulnerabilities...
58 12:02:56  WARN  This OS version is no longer supported by the distribution: alpine 3.3.3
59 12:02:56  WARN  The vulnerability
      detection may be insufficient because security updates are not provided
```

Ferramentas SCA fazem análise estática sobre pacotes binários do software. Nesse sentido, outro tipo de análise estática não explorado neste texto é a análise de binários de aplicativos móveis com auxílio de ferramentas. O MobSF (`https://github.com/MobSF/Mobile-Security-Framework-MobSF`) é uma ferramenta de análise de vulnerabilidades em aplicativos móveis capaz de fazer análises estáticas e dinâmicas sobre os binários dos aplicativos em vários formatos (por exemplo, APK, XAPK, IPA e APPX). Testes de segurança em aplicativos móveis já foram tratados em outro minicurso [Braga et al. 2012].

5.3.4. Testes de segurança auxiliados por ferramentas DAST

Esta seção trata ferramentas DAST livres e gratuitas, tais como, por exemplo, OWASP ZAP `https://www.zaproxy.org/`) e Burp Community (`https://portswigger.net/burp`). O texto consiste em testes de segurança manuais apoiados pelas ferramentas DAST, com base em catálogos de vulnerabilidades conhecidas, como OWASP Top 10 [top 2021] e CVEs conhecidos (`https://cve.mitre.org/`). O OWASP ZAP também é usado em varreduras de vulnerabilidades.

5.3.4.1. Testes de segurança com OWASP ZAP

O OWASP ZAP `https://www.zaproxy.org/`) é uma ferramenta livre e de código aberto, para a realização de análise dinâmica e projetada para identificar vulnerabilidades em aplicações web. Ele oferece recursos abrangentes para avaliar a segurança de um aplicativo, incluindo a detecção de falhas de segurança comuns, como injeção de SQL, *cross-site scripting* (XSS) e configurações incorretas. Com suporte para testes automatizados e manuais, o OWASP ZAP é amplamente utilizado por desenvolvedores e profissionais de segurança para análises dinâmicas em aplicações e testes de segurança em APIs.

Os comandos de linha mostrados na listagem 5.14 fazem as varreduras de vulnerabilidades com OWASP ZAP nas quatro aplicações web vulneráveis usadas neste texto. A linha de comando o ZAP contém os seguintes elementos:

- `zap.sh` é o *script* de ativação do ZAP.
- `-cmd` indica execução por comando de linha.
- `-quickurl http://<<EXEMPLO.COM>>` informa a URL alvo da varredura.
- `-quickprogress` mostra uma barra de progresso em texto.
- `-quickout <<arquivo>>` informa arquivo de saída. A extensão indica o tipo do arquivo (.html, .json, .md, .xml).

A tabela 5.3 mostra resumidamente os alertas emitidos pelo ZAP para cada uma das aplicações analisadas. A varredura detectou problemas em 4 níveis de alertas: alto, médio, baixo e informacional (vazamento de informação). Apenas as aplicações Gruyere e Juice Shop possuem alertas altos. Em quantidade de alertas, a distribuição fica em Gruyere com 123 alertas, bWAPP com 409 alertas, WebGoat com 171 alertas e Juice Shop com 370 alertas, fazendo do bWAPP a aplicação mais vulnerável desta análise com o ZAP. Claro, exceto pelos possíveis falsos positivos, os alertas são verdadeiros.

Listagem 5.14. Varreduras de vulnerabilidades com OWASP ZAP.

```
1
2 $ zap.sh -cmd -quickurl http://localhost:3000 -quickprogress -quickout ./zap-juice.html
3
4 ...
5
6 $ zap.sh -cmd
      -quickurl http://localhost:8080/WebGoat -quickprogress -quickout ./zap-webgoat.html
7 ...
8
9 $ zap.sh -cmd -quickurl http://localhost:8090 -quickprogress -quickout ./zap-bwapp.html
10
11 ...
12
13 $ zap.sh -cmd -quickurl ttp://localhost:8008 -quickprogress -quickout ./zap-gruyere.html
```

O OWASP ZAP foi utilizado até este ponto como uma ferramenta de linha de comando para varredura de vulnerabilidades. Porém, ele é muito mais que isso. O ZAP também possui uma interface gráfica (GUI) e um dos seus usos mais importantes é como *proxy* de aplicação (ou

Tabela 5.3. Alertas emitidos pelo ZAP durante as varreduras das aplicações vulneráveis.

Aplicação	Alerta	Nível de Risco	Quant.
Gruyere	Cross Site Scripting (Refletido)	Alto	2
Gruyere	Metadados de nuvem potencialmente expostos	Alto	1
Gruyere	Ausência de tokens Anti-CSRF	Médio	3
Gruyere	Content Security Policy (CSP) Header Not Set	Médio	10
Gruyere	Hidden File Found	Médio	4
Gruyere	Injeção CRLF	Médio	1
Gruyere	Missing Anti-clickjacking Header	Médio	10
Gruyere	Cookie No HttpOnly Flag	Baixo	1
Gruyere	Cookie without SameSite Attribute	Baixo	1
Gruyere	Server Leaks Version Information via HTTP Response Header	Baixo	11
Gruyere	X-Content-Type-Options Header Missing	Baixo	11
Gruyere	Cookie Poisoning	Informativo	2
Gruyere	Cookie com Escopo Fraco	Informativo	1
Gruyere	Divulgação de Informações - Comentários Suspeitos	Informativo	1
Gruyere	Modern Web Application	Informativo	4
Gruyere	User Agent Fuzzer	Informativo	60
bWAPP	Application Error Disclosure	Médio	27
bWAPP	Ausência de tokens Anti-CSRF	Médio	5
bWAPP	Content Security Policy (CSP) Header Not Set	Médio	36
bWAPP	Hidden File Found	Médio	1
bWAPP	Missing Anti-clickjacking Header	Médio	35
bWAPP	Navegação no Diretório	Médio	32
bWAPP	Cookie No HttpOnly Flag	Baixo	2
bWAPP	Cookie without SameSite Attribute	Baixo	2
bWAPP	Vaza informações via HTTP Response Header X-Powered-By	Baixo	10
bWAPP	Server Leaks Version Information via HTTP Response Header	Baixo	81
bWAPP	X-Content-Type-Options Header Missing	Baixo	75
bWAPP	Cookie com Escopo Fraco	Informativo	2
bWAPP	Divulgação de Informações - Comentários Suspeitos	Informativo	1
bWAPP	GET for POST	Informativo	1
bWAPP	User Agent Fuzzer	Informativo	96
bWAPP	User Controllable HTML Element Attribute (Potential XSS)	Informativo	3
WebGoat	Ausência de tokens Anti-CSRF	Médio	3
WebGoat	Content Security Policy (CSP) Header Not Set	Médio	4
WebGoat	Missing Anti-clickjacking Header	Médio	3
WebGoat	Session ID in URL Rewrite	Médio	1
WebGoat	Cookie without SameSite Attribute	Baixo	2
WebGoat	Server Leaks Version Information via HTTP Response Header	Baixo	14
WebGoat	X-Content-Type-Options Header Missing	Baixo	7
WebGoat	Cookie com Escopo Fraco	Informativo	2
WebGoat	Divulgação de Informações - Comentários Suspeitos	Informativo	3
WebGoat	User Agent Fuzzer	Informativo	132
Juice Shop	Metadados de nuvem potencialmente expostos	Alto	1
Juice Shop	Configuração Incorreta Entre Domínios	Médio	85
Juice Shop	Content Security Policy (CSP) Header Not Set	Médio	70
Juice Shop	Cross-Domain JavaScript Source File Inclusion	Baixo	124
Juice Shop	Divulgação de Data e Hora - Unix	Baixo	1
Juice Shop	Divulgação de Informações - Comentários Suspeitos	Informativo	2
Juice Shop	Modern Web Application	Informativo	63
Juice Shop	User Agent Fuzzer	Informativo	24

proxy de navegador web). Os próximos parágrafos mostram resumidamente como configurar o OWASP ZAP e em seguida usá-lo para testar manualmente a segurança de aplicações web.

Após instalado com sucesso, o ZAP pode ser iniciado pelo ícone na área de trabalho ou pelo *script* de linha de comando (zap.sh ou zap.bat). A GUI do ZAP inicia com uma janela de abertura, progresso e anúncios (até a versão 2.12.0).

- Em seguida, o ZAP solicita que o usuário escolha como deseja persistir a sessão.
- Neste treinamento, prefere-se a opção Não.
- Clicar no botão de Início.

A tela inicial (início rápido) do ZAP oferece três maneiras de iniciar:

- Varredura automatizada (*Automated Scan*).
- Exploração manual (*Manual Explore*).
- Documentação (*Learn more*).

A seguir, as funcionalidades de varredura automática e exploração manual são demonstradas sobre as aplicações vulneráveis para aprendizado Juice Shop e WebGoat 7.1 (já apresentadas anteriormente). As aplicações vulneráveis devem ser iniciadas antes de prosseguir com os exercícios. A função *Automated Scan* equivale à varredura em comando de linha e pode ser ativada do seguinte modo:

- Clicar no botão de *Automated Scan*.
- Na Janela *Automated Scan*.
  - Fornecer a URL a ser varrida, p.ex. `http://localhost:8080/WebGoat`.
  - Usar as opções *Spider*, *Ajax Spider* com *Firefox headless*.
  - Clicar no botão *Attack*/Ataque.
- O progresso da varredura é mostrado na barra de progresso do *Spider/AjaxSpider*.
- Os ataques automáticos são mostrados na aba Varredura Ativa.
- O resultado do ataque é mostrado na aba Alertas.
- A varredura pode alertar sobre 4 níveis de problemas: alto, médio, baixo e informacional (vazamento de informação).

O teste automático de outra aplicação mostra resultado diferente? Para verificar esta hipótese, basta repetir o procedimento anterior com uma URL diferente. Por exemplo, a URL do Juice Shop (`http://localhost:3000`).

A exploração manual (*Manual Explore*) segue o seguinte procedimento inicial:

- Na Janela Início Rápido, clique no botão de Manual Explore.
- Na janela Manual Explore.
  - Fornecer a URL a ser explorada, p.ex. `http://localhost:8080/WebGoat`.
  - Desmarcar a caixa *Enable HUD*.
  - Escolher o *browser* a ser aberto: Firefox ou Chrome.
  - Ativar o botão *Launch Browser*.

A exploração manual é a atividade de teste de segurança manual propriamente dita e utiliza-se, conforme a necessidade do testador, de diversos procedimentos, tais como criação de *breakpoints* e filtros, intercepção e modificação de requisições, edição e reenvio de requisições e o uso de opções de *proxy*. Estes procedimentos são detalhados da seguir.

A criação de *breakpoints* e filtros é útil para a interceptação apenas das requisições úteis para o trabalho de teste. Por exemplo, utiliza-se o seguinte procedimento para criar um *breakpoint* para as requisições HTTP do tipo POST:

- Clicar no botão X vermelho, *breakpoint*, na barra de botões de controle de *breakpoint* do ZAP (a barra com botões de ativa/desativa, avançar, *play next*, e *breakpoint*).
- Na janela aberta de criação de *breakpoints*, preencher os campos:
  - *String*: POST.
  - Localização: Cabeçalho da Solicitação (*Request Header*).
  - *Match*: Contém / *Contain*.
  - Clicar no botão Salvar.

Os *breakpoints* criados aparecem na aba ou guia de *breakpoints* e podem ser ativados/-desativados pelo *checkbox*. Assim o controle manual de *breakpoint* (botões verde/vermelho) não é mais necessário.

Já o filtro de requisições para o WebGoat pode ser configurado do seguinte modo. Na guia de histórico de requisições, clicar no botão de Filtro, inserir no campo URL *Inc Regex* a *string* `.*.WebGoat.*.` (incluindo os pontos) para mostrar apenas as requisições relacionadas ao WebGoat.

A intercepção e modificação de requisições pode ser experimentada com o seguinte procedimento (com o *breakpoint* ativado):

- No WebGoat 7.1, ir para o menu General → Http *basics*.
- No campo de formulário *Enter your name*, digite a *string* “teste” e ative o botão *Go!*.
- O ZAP indicará que interceptou a requisição.
- No corpo da requisição, modificar o valor do parâmetro como quiser.

- Por exemplo, adicionando "123" ao final da *string* "teste".
- Clicar em um dos botões *Play Next* ou Avançar para dar andamento à requisição.
- (Lembrar de desativar o *breakpoint*!)

A edição e reenvio de requisições podem ser experimentadas com facilidade. No Histórico de requisições do ZAP, selecionar a requisição mais recente e seguir o procedimento:

- Clicar com botão direito do mouse, escolher a opção Reenviar.
- O ZAP abre a janela editor manual de requisições.
- Na janela de edição de requisição:
  - Na guia Requisição modificar o parâmetro no corpo.
  - Na guia Resposta, que está vazia ainda, clicar no botão Enviar.
  - A resposta é enviada.
  - Lembrar de fechar a janela do editor manual de requisições.

As opções de *proxy* podem ser experimentadas com o seguinte procedimento. O ZAP pode servir de intermediário entre o navegador e a aplicação web. Há três maneiras de fazer isto:

- A maneira antiga, configurar manualmente o ZAP como *proxy* de aplicação no navegador. Esta opção não será usada neste treinamento.
- Por meio da opção de início rápido Manual Explore, como visto anteriormente.
- Com o botão de ativação do navegador (ícone do navegador), abrir um navegador intermediado e controlado remotamente pelo ZAP *Proxy*.

Os procedimentos explicados até aqui são usados em dois exemplos de exploração de vulnerabilidades. O exemplo a seguir mostra a exploração de uma vulnerabilidade de tratamento incorreto de erro no WebGoat com OWASP ZAP:

- No WebGoat 7.1 via ZAP Proxy.
  - Menu *Improper Error Handling* → *Fail Open Authentication Scheme*.
  - No formulário, digitar user/password administrativos (`webgoat/webgoat`).
  - clicar no botão Login.
  - Logou como admin? Fazer logout.
- No ZAP, ativar o *breakpoint* de interceptação de requisições.
- No WebGoat, fazer novamente o login *webgoat/webgoat* no mesmo formulário.
- O ZAP indica que interceptou a requisição, remover o parâmetro senha do corpo da requisição.

  - Era assim: `Username=webgoat&Password=webgoat&SUBMIT=Login`.
  - Fica assim: `Username=webgoat&SUBMIT=Login`.
- Enviar a requisição (Botão *Play* dos controles de *breakpoint*).
- Terá logado como admin sem fornecer a senha!
- Erro de projeto de programa! WebGoat falha em aberto em caso de erro na autenticação.

O Exemplo de exploração a seguir contorna ou burla a validação de dados de entrada feira pelo lado cliente (*frontend*) da aplicação.

- Entrar no seu deploy do WebGoat `http://localhost:8080/WebGoat/`.
- Logar na aplicação como usuário/senha: guest/guest
- Selecionar a opção de menu *Parameter Tampering* → *Exploit Hidden Fields*.
- Testar a funcionalidade de *Shopping Cart* (Carrinho de Compras).
  - Editar quantidade, campo *Quantity*;
  - Atualizar o carrinho, botão *UpdateCart*;
  - Realizar a compra, botão *Purchase*.
- Qual o comportamento do valor total da compra? Será possível editá-lo?
- Abrir as ferramentas de desenvolvedor do seu navegador.
- Com a ferramenta de seleção (ponteiro), selecionar o FORM do carrinho de compras.
- Encontrar no FORM a campo *"Price"* do tipo HIDDEN.
- No código do FORM para o campo *"Price"*, editar o valor 2999.99 para 0.99.
- Na aplicação, ativar o botão *UpdateCart*.
- O que aconteceu? Qual o valor final da compra?
- Isto é uma vulnerabilidade de código ou uma falha de projeto?

5.3.4.2. Testes de segurança com *Burp Community*

Esta subseção mostra o funcionamento da ferramenta de testes de intrusão *Burp Suite Community* (`https://portswigger.net/burp`) para auxiliar a descoberta e a exploração de vulnerabilidades presentes nas aplicações avaliadas, em particular, o bWAPP. O *Burb Suite Community Edition* pode ser baixado de `https://portswigger.net/burp/communitydownload`. Baixar a versão apropriada para o sistema operacional e que seja a mais recente e estável. Seguir o processo de instalação conforme instruções do programa. O procedimento a seguir realizar as configurações iniciais do Burp:

- Ativar a ferramenta Burp como desejar (ícone ou comando de linha).
- *Burp Community* só aceita projetos temporários, na janela inicial, clicar no botão *Next*.
- Usar as opções *default*. Clicar no botão *Start Burp*.
- Observar que nada acontece na aba (ou guia) *Target*.
- Ir para a aba *Proxy*.
- Burp possui um *browser* embarcado e configurado como *proxy*, botão *Open browser*.
- Na guia Proxy → Intercept, desativar o controle *Intercept is on* (de on para off)
- Na janela do *browser*, digitar a URL ou IP do bWAPP (que deve estar ativado).
  - Selecionar a aplicação bWAPP, logar na aplicação com usuário/senha `bee/bug`.
- No Burp, guia *Target* → *site map*, abrir a pasta bWAPP e a engrenagem (login.php).
  - Observar que as informações de login e senha estão expostas!
  - Clicar com botão direito em bWAPP e selecionar a opção *Add to scope* e apertar o botão *Yes*.
- Clicar na barra *Filter* e selecionar a opção *Show only in-scope items*.

As configurações do Burp foram realizadas com sucesso. Como o ZAP, o Burp também possui funcionalidades que auxiliam os testes manuais: opções de *proxy*, *Repeater* e *Intruder*. A opções de *proxy* do Burp podem ser configuradas do seguinte modo:

- Acessar a aba Proxy → Options.
  - Na seção *Intercept Client Requests*, desativar a regra *File extension Does not match*.
  - Na seção *Intercept Server Responses*, ativar as regras de interceptação e ativar a regra *Content type header Matches text*.
  - Na guia *Proxy* → *Intercept*, ativar o controle *Intercept is on* (de *off* para *on*).
- Testar o comportamento destas modificações da interceptação da bWAPP.
- Em *Proxy* → *Options*, na seção *Response Modification*, ativar a opção *Unhide hidden form fields* e a opção *highlight unhidden fields*.
- Na guia *Proxy* → *Intercept*, desativar o controle *Intercept is on* (de *on* para *off*).
- Testar o comportamento destas modificações da interceptação da bWAPP.
  - Selecionar o *bug Insecure DOR (Change Secret)* e clicar no botão *Hack*.
  - Um campo *hidden* aparecerá na janela do *browser*.

Repeater é uma ferramenta muito útil em testes manuais. Ele permite a manipulação (alteração) de parâmetros das requisições individuais para a repetição de uma requisição com parâmetros personalizados pelo testador. Para usar o *Repeater*:

- Na aba *Proxy* → HTTP *History*, achar a requisição de login da aplicação bWAPP.
- Clicar na requisição com o botão direito do mouse e selecionar a opção *Send to Repeater*.
- O rótulo da aba *Repeater* vai mudar de cor para vermelho, indicando que foi ativada.
- A aba *Repeater* mostra a requisição que será repetida.
- Ativar o botão *Send* para enviar a requisição como está.
- Visualizar a resposta obtida (o login foi bem-sucedido).
- Editar o campo `password=bug` para `password=bag` e ativar o botão *Send*.
- Visualizar a resposta, o botão *Render* para ver em HTML (o login foi mal-sucedido).

Intruder é uma ferramenta muito útil em testes de força bruta. Ele permite a manipulação (alteração) de parâmetros das requisições e o envio de diversas requisições personalizadas em massa. Para usar o *Intruder*:

- Na aba *Proxy* → HTTP *History*, achar a requisição de login da aplicação bWAPP.
- Clicar na requisição com o botão direito do mouse e selecionar a opção *Send to Intruder*.
- O rótulo da aba *Intruder* vai mudar de cor para vermelho, indicando que foi ativado.
- Na aba *Intruder* → *Positions*, manter *attack type* em *Sniper*.
  - Observar todos os campos que podem ser manipulados. Clicar no botão *Clear §*.
  - Selecionar o parâmetro *bug*, clicar no botão *ADD §*, o parâmetro fica assim: *§bug§*.
- Na aba *Intruder* → *Payloads*, manter o *Payload type* em *Simple list*.
  - Na caixa de *Payload options*, adicionar (ADD) os itens da lista manualmente.
  - Digitar cada item da lista bag, beg, big, bog, bug seguido de enter.
  - `bag <ENTER>beg<ENTER>big<ENTER>bog<ENTER>bug`.
  - Clicar no botão *Start Attack* no canto superior direito da janela do Burp.
  - O progresso do ataque é mostrado em outra janela.

Os procedimentos explicados até aqui são usados em exemplos de exploração de vulnerabilidades. O procedimento a seguir mostra a exploração de HTML *Injection* no bWAPP com Burp. Ainda na tela do *bWAPP* em *proxy* pelo *Burp*.

- Selecionar o bug *HTML Injection - Reflected (POST)*.

- Preencher os campos de nomes e sobrenome e clicar em *Go!*,
  - p.ex., preencher com "Tony"e "Stark".
  - O conteúdo fornecido pelo usuário é mostrado no navegador.
- No *Burp*, *Proxy* → *Intercept*, ativar a Interceptação (*Intercept is on*).
- No *browser*, preencher novamente com "Tony"e "Stark". Clicar em *Go*.
- No *Burp*, verificar que a requisição interceptada.
  - Observar os parâmetros `firstname=Tony&lastname=Stark&form=submit`.
  - Substituir a linha de parâmetros por `firstname=<h1><a href="http://www.google.com">Clique</a></h1> &lastname=<h2>Hahah!</h2>&form=submit`.
  - Clicar no botão *Forward* enquanto houver requisições para tratar.
- No *Browser*, observa-se que o HTML foi inserido na página e apresentado ao usuário.

Outra injeção de HTML via *GET* no bWAPP:

- Selecionar o bug *HTML Injection - Reflected (GET)*.
- Preencher os campos de nomes e sobrenome e clicar em *Go!*,
  - p.ex., preencher com "Tony"e "Stark".
  - O conteúdo fornecido pelo usuário é mostrado no navegador e na URL também!
- No *Burp*, *Proxy* → *Intercept*, ativar a Interceptação (*Intercept is on*).
- No *browser*, preencher novamente com "Tony"e "Stark". Clicar em *Go*.
- No *Burp*, verificar que a requisição interceptada
  - Observar os parâmetros `firstname=Tony&lastname=Stark&form=submit`.
  - Substituir a linha de parâmetros por `firstname=<h1><a href="http://www.google.com">Clique</a></h1> &lastname=<h2>Hahah!</h2>&form=submit`.
  - Clicar no botão *Forward* enquanto houver requisições para tratar.
- No *Browser*, observa-se que o HTML foi inserido na página e apresentado ao usuário.

Mais uma injeção de HTML via *GET* no bWAPP:

- Selecionar o bug *HTML Injection - Reflected (URL)*.
  - Não esquecer de clicar em *Hack!*
  - A página mostra a URL atual (*Current URL*)

- No *Burp*, *Proxy* → *Intercept*, ativar a Interceptação (*Intercept is on*).
- No *browser*, recarregar a página (F5 ou botão de recarregar no navegador).
- No *Burp*, verificar que a requisição interceptada
  - Substituir o endereço IP do *header Host* por algum outro endereço.
    - P.ex., *www.google.com*
  - Clicar no botão *Forward* enquanto houver requisições para tratar.
- No *Browser*, observa-se que a URL atual (*Current URL*) foi adulterada.

O próximo exemplo explora um XSS JSON no *bWAPP* em *proxy* pelo *Burp*.

- Selecionar o bug *XSS - Reflected (JSON)*.
- Testar a busca com qualquer nome de filme, p.ex., *"Matrix"*.
  - Observar que o conteúdo do campo de busca volta para o usuário;
    - *"Matrix"*?
- No *Burp*, em *Proxy* → *HTTP History*, achar a última requisição com a busca *"Matrix"*.
  - Observar o conteúdo da resposta (*Response*). Onde está a palavra *"Matrix"*?
    - Dica: Usar a função de busca na parte inferior da baixa da *Response*.
  - A palavra buscada (*"Matrix"*) está inserida entre *tags* `<script> ... </script>`
    - O JSON de resposta é construído no trecho.
    - Um *Eval* vulnerável também está exposto.
- O ataque consiste em customizar um JSON de resposta com um *script* embutido.
  - Finalizar o JSON, fechar o *script* e inserir o *script* malicioso.
  - Observar de onde o JSON é finalizado e copiar os caracteres finais
    - Dica: a linha do JSON termina com `"}]}';`
  - Fechar o *script* do JSON e inserir o *script* malicioso
    - `"}]}'; </script><script>alert(1);</script>`
  - Outro exemplo de *script* malicioso:
    - `"}]}';</script><script>alert(document.cookie);</script>`

Ainda no *bWAPP* em *proxy* pelo *Burp*, o próximo exemplo explora uma *SQL Injection*.

- No *bWAPP*
  - Selecionar o bug *SQL Injection (POST/Select)*.
  - Escolher um filme da lista de seleção e clicar em *Go*.

- No *Burp*, *Proxy* → *Proxy History*, selecionar a requisição mais recente.
  - Observar como o parâmetro de busca *movie* é usado.
  - Copiar a requisição para o *Repeater*.
- No *Burp Repeater*
  - Modificar *movie* da requisição para `movie=1 or 1=1\#&action=go`.
  - Clicar em *Send*. O que acontece? Dica: o comportamento não foi modificado.
  - Modificar *movie* da requisição para *movie=200 union select 1,2,3,4,5,6,7#*
  - Clicar em *Send*. O que aconteceu dessa vez?

5.4. Falhas de projeto de software e segurança de APIs

Esta seção relaciona vulnerabilidades de API às falhas de projeto de software, discutindo as falhas de projeto de software mais comuns (*Top 10 Software Design Flaws*) e seu impacto na segurança das APIs. A seção contempla os testes de segurança de APIs REST auxiliados por ferramentas DAST (p.ex., OWASP ZAP) e SAST (p.ex., SonarQube, HorusSec). A seção também aborda duas classificações bastante conhecidas de ameaças e ataques mais comuns contra APIs: *OWASP Top 10 API Risks* (edição de 2023) e maus usos de criptografia relacionados a *JSON Web tokens* (JWT).

5.4.1. Falhas de projeto de software

De acordo com [iee 2014], existem 10 falhas de design de software mais comuns que devem ser evitadas quando do projeto de uma arquitetura. As próximas seções detalham cada uma delas.

5.4.1.1. Ganhar ou criar confiança, nunca supor que ela já existe

Em uma aplicação, cliente-servidor (*frontend - backend*), implementar no cliente as funções de segurança do servidor expõe estas funções ao ambiente menos confiável do cliente ou *frontend*. Designs que colocam essas funções de segurança no cliente são inerentemente fracos e inseguros. Por exemplo, dados confidenciais de autorização, controle de acesso, verificação de regras de segurança e lógica de negócios ficam vulneráveis. Existem vários exemplos de clientes pouco confiáveis, tais como: navegadores de internet, clientes ricos (*Thick/Fat/Rich clients*), dispositivos móveis, software embarcado e chamadas de APIs por parceiros externos. Dados enviados por clientes não confiáveis são considerados comprometidos até prova do contrário e, por isso, devem ser validados antes de usados.

Essa falha de design geralmente acontece quando projetistas fazem o seguinte (incorretamente): assumem que APIs do servidor sempre serão chamadas na mesma ordem, acreditam que a interface do usuário é sempre capaz de restringir o que o usuário envia para o servidor, tentam construir a lógica de negócios ou validações exclusivamente no lado do cliente ou tentam armazenar segredos no cliente, mesmo usando criptografia forte.

5.4.1.2. O mecanismo de autenticação não pode ser desviado nem adulterado

Autenticação é o ato de validar a identidade de algo ou alguém. Um software seguro deve impedir que um usuário não autenticado tenha acesso aos sistemas e serviços. Depois que o usuário é autenticado, o sistema seguro também deve impedir que o usuário altere sua identidade sem nova autenticação. Em geral, um sistema deve considerar a força da autenticação que um usuário forneceu antes de agir, a qual depende do fator de autenticação usado: senha, *token*, biometria, ou uma combinação de dois deles. Por exemplo, autenticação por *token* de sessão mantido em *cookie* poder ser suficiente em alguns casos, mas não em todas as ocasiões.

Um sistema que possui um mecanismo de autenticação é vulnerável ao desvio de autenticação quando permite que um usuário acesse o serviço / API diretamente por um *token* ou URL "escondida", sem exigir uma credencial de autenticação obtida por meio do processo de autenticação normal. Em geral, é preferível ter um único método, componente ou sistema responsável pela autenticação de usuários porque isto evita inconsistências entre réplicas das bases de usuários. Uma vez que o mecanismo de autenticação tenha sido escolhido, deve ser utilizado em todas ações de autenticações. Por outro lado, tal mecanismo único pode funcionar como um gargalo lógico que não pode ser contornado, sendo um ponto de falha único e, por isso, sujeito a ataques de negação de serviço.

5.4.1.3. Autorizar somente após autenticar (Autorização != Autenticação)

A autorização sempre deve ser feita por verificação explícita de uma permissão, mesmo depois que uma autenticação acabou de ser realizada. A autorização depende dos privilégios do usuário autenticado e também do contexto da solicitação (por exemplo, horário, lugar, equipamento, etc.). Há casos em que a autorização (de um usuário para um sistema ou serviço) precisa ser revogada. Se a infraestrutura de autorização não permitir a revogação, o sistema é vulnerável aos abusos de usuários com autorizações desatualizadas. Para operações confidenciais, sensíveis ou críticas, a autorização pode exigir ainda a reautenticação do usuário.

Autenticação não é binária, porque os usuários podem ser obrigados a apresentar (novas) evidências (mais fortes) de sua identidade. Além disso, a autenticação geralmente não é contínua porque acontece a intervalos periódicos. Por exemplo, entre os momentos de (re)autenticação, um usuário pode estar autenticado, mas se afastar do dispositivo ou entregá-lo para outra pessoa. A autorização de uma operação sensível pode exigir reautenticação ou autenticação mais forte e políticas de segurança podem exigir duas ou mais autorizações em ações críticas (obedecendo ao princípio de separação de responsabilidades). Assim como ocorre com a autenticação, uma infraestrutura comum (um único mecanismo) deve ser usada para as verificações de autorização, a fim de evitar inconsistências, mas com a desvantagem do ponto único de falha.

5.4.1.4. Separar controle e dados, não executar comandos não confiáveis

Misturar dados e instruções de controle em uma única estrutura (por exemplo, strings em requisições HTTP), pode levar a vulnerabilidades de injeção. Falta de separação estrita entre dados e comandos/controles resulta em dados não confiáveis. O atacante manipula os dados para controlar o fluxo de execução de uma aplicação. Em software escrito em linguagens de mais baixo

nível, como C, C++, *assembly*, ou software embarcado, a mistura entre dados e controle pode causar corrupção de memória e pode levar a ataques de negação de serviço, vazamentos de dados e execução de código malicioso. Em linguagens de alto nível, a mistura de controle e dados pode ocorrer na interpretação de comandos em tempo de execução das linguagens de programação.

Software que ignora o princípio da separação estrita entre dados e código, sem separação explícita entre dados e instruções, são inerentemente inseguros. Esta falha de projeto geralmente ocorre quando software monta strings concatenando dados não confiáveis e instruções de controle confiáveis. Vulnerabilidades de injeção surgem quando os dados não confiáveis não são validados nem jogados fora. APIs propensas a injeção de comandos possuem risco alto de exploração da vulnerabilidade de injeção. Exemplos de tais vulnerabilidades incluem injeção de SQL, injeção de JavaScript (em ataques XSS), injeção de XML e injeção de comando do sistema operacional.

5.4.1.5. Garantir que todos os dados são validados explicitamente

É importante garantir que todos os dados são validados explicitamente. Uma boa pática, neste caso é usar validadores centralizados para validar todos os dados de entrada do mesmo jeito. O validador é um filtro ou interceptador na arquitetura da aplicação. Ele converte dados externos para um formato interno, antes de validações sintáticas e semânticas.

Protocolos de comunicação devem validar todos os campos de todas as mensagens recebidas (antes de processá-las). Arquivos de dados em formatos complexos (XML, imagens, textos ricos, etc.) devem passar por validações sintáticas (de formato e estrutura) e semânticas (da lógica de negócios). Na validação sintática devem ser usadas bibliotecas de validação que reconhecem formatos estruturados, como, por exemplo, e-mail, URLs, CPFs. Além disto, tipos de dados estruturados devem ser usados para capturar suposições sobre validade de dados. Por exemplo, *string* que representa data/hora deve ser validada se contém uma data/hora bem formada (p.ex., `DD/MM/YYYY`). Já na validação semântica, a validação de entrada geralmente depende do estado da aplicação (da lógica de negócios) e consiste em reavaliar suposições, premissas e pré-condições sobre os dados, nos trechos de código próximos ao uso dos dados.

Nas APIs, os *endpoints* (as funções da API) que consomem strings com controle e dados ao mesmo tempo devem ser evitados. Neste casos, é preferível expor *endpoints* que consumam tipos estruturados (com segregação estrita entre dados e comandos/controle), como por exemplo, os dados com tipos fortes: inteiros, booleanos e alfabéticos. Nos aplicativos, APIs que misturam dados e *flags* de controle em parâmetros de strings também devem ser evitados. Se um aplicativo deve usar uma API legada (propensa a injeção), esta API deve ser acessada por meio de uma API interna (um *proxy* ou *wrapper* de API) com segregação de dados e controle. Nos aplicativos que transformam dados em códigos, os dados devem ser validados com rigor ou cuidado maior e as ações executadas com os comandos gerados devem ser limitadas e âmbito restrito.

Comandos como `eval` ou `exec` são comuns nas linguagens interpretadas, como Java, Python, Ruby e JavaScript. Estes comandos consomem uma cadeia de caracteres (*string*) e invocam o interpretador da linguagem ou executam um comando do sistema operacional. Se o uso dos comandos `exec` ou `eval` são inevitáveis, uma API intermediária (*wrapper* de API) deve ser colocada entre o código da aplicação e as interfaces de uso do `eval` ou `exec`, enquanto só a API mais segura é exposta para a aplicação. Em programação orientada a objetos,

reflexão computacional é um recurso poderoso de introspecção de programas e alteração do código ou do comportamento de uma aplicação em tempo de execução. O uso de reflexão computacional é escolha de design arriscada porque defeitos podem levar a execução de código malicioso, como por exemplo, inclusão ou modificação de métodos/funções em um objeto.

5.4.1.6. Usar criptografia corretamente

É um fato conhecido que a maioria das vulnerabilidades de criptografia está no uso incorreto de bibliotecas criptográficas e não em vulnerabilidades do código criptográfico em si [Braga and Dahab 2016, Braga and Dahab 2017]. Os maus usos ou usos incorretos de criptografia por desenvolvedores de software estão fora do escopo deste texto e já foram tratados detalhadamente em minicursos anteriores [Braga and Dahab 2015b, Braga and Dahab 2018, Braga and Dahab 2019].

Uma vez que implementações criptográficas de qualidade e boa reputação estão disponíveis para todos, a fonte mais comum de problemas com criptografia é o software em torno da criptografia, escrito por desenvolvedores não especialistas em criptografia nem em segurança. Os falhas de projeto associadas ao uso incorreto de criptografia são as seguintes: não prever adaptação ou evolução da criptografia, usar ou inventar sua própria criptografia, usar incorretamente bibliotecas ou APIs criptográficas, gestão (incluindo geração, distribuição, armazenamento) inadequada de chaves criptográficas, gerar aleatoriedade insuficiente ou falsa e usar diversas implementações criptográficas diferentes e descentralizadas.

5.4.1.7. Identificar os dados sensíveis e como eles devem ser mantidos

Só é possível proteger o que se conhece, por isso é importante identificar os dados sensíveis para saber como protegê-los melhor. Projetistas devem identificar dados confidenciais ou privados das suas aplicações e determinar como protegê-los. Neste casos, uma avaliação com base na Lei Geral de Proteção de Dados (LGPD) pode ser utilizada. O software deve proteger os dados sensíveis nas diversas situações, sejam dados em repouso (armazenados), em trânsito e em uso. A sensibilidade dos dados e as proteções decorrentes dela podem mudar com o tempo porque negócios evoluem, regulamentações mudam, sistemas são interconectados e novas fontes de dados são incorporadas à aplicação.

A sensibilidade dos dados tratados pela aplicação depende do contexto de uso específico (legislação, política, contratos, vontade do usuário, etc.). Uma política de classificação em níveis orienta o manuseio seguro dos dados. Um exemplos bastante comum de níveis de classificação é a escala de acesso a informação em público, restrito, privado, secreto e ultra secreto. As aplicações em software geralmente tratam muitos tipos de dados sensíveis, que podem ser fornecidos pelo usuário, derivados pela aplicação, coletados de sensores, material criptográfico, informações bancárias, de compras, de cartões e informações privadas (p. ex., PII), entre outros.

5.4.1.8. Sempre considerar a criatividade do usuário final

Muitos usuários são capaz de vislumbrar jeitos diferentes de usar a aplicação. A segurança da aplicação depende do que os usuários fazem com ela, por isso é importante sempre considerar como os usuários usam a aplicação. Os usuários legítimos de um software variam desde o pessoal da implantação, operação, configuração e manutenção até os usuários finais. Há tanto usuários legítimos sem interesse em segurança quanto usuários legítimos e mal-intencionados. Além do usuário final que usa a aplicação por meio de uma interface, os programadores que usam APIs também podem ser atacantes.

Usabilidade e experiência do usuário são geralmente importantes para a operação segura do software porque o usuário não se sente compelido a burlar controles de segurança difíceis de usar. De modo geral, o software deve ser configurado e usado com segurança por meio de interfaces intuitivas e fáceis de usar e controles de segurança expressivos e sem excesso. Quando a segurança é muito difícil de configurar para uma grande população de usuários, ela nunca será configurada para todos ou não será configurada corretamente, permanecendo incompleta.

Ataques de escalada de privilégios podem resultar de falhas ou falta de autorização explícita. Uma falha comum ocorre quando a autorização não está vinculada ao usuário autenticado. Isto é, o software supõe erroneamente que o usuário não tem acesso ou tem acesso, sem uma verificação explícita. A falta de compartimentação e isolamento entre usuários é outro problema comum em que um usuário acessa dados de outro usuário. Além disso, configurações "abertas"favorecem navegação “livre” do usuário legítimo e mal intencionado.

Muitas vezes existem decisões de projeto que exigem um compromisso ou equilíbrio entre segurança e usabilidade. “Danos colaterais” a usabilidade podem ocorrer na implementação de segurança na interface do usuário e vice-versa. Por exemplo, dificultar o acesso para evitar vazamentos de dados em interfaces acessíveis, ou facilitar o *shoulder surfing* em aplicativos móveis usados pelos passageiros do transporte público.

Muitas vezes, vulnerabilidades aparecem de situações raras e exceções. Por exemplo, ao não considerar o apagamento e a revogação do usuário ao final do seu ciclo de vida, por exemplo em uma rescisão contratual ou troca de função. Outro exemplo seria não considerar requisitos de segurança específicos para classes diferentes de usuários com necessidades específicas de usabilidade (tais como PCDs, proficiência em idiomas, crianças, idosos, pessoas com diferentes capacidades cognitivas) em softwares utilizados pelo público em geral.

5.4.1.9. Os componentes externos mudam a superfície de ataque

Atualmente, o uso de *frameworks* de software, bibliotecas externas ou outros componentes de terceiros é prática comum no desenvolvimento de software. Porém, a integração ou dependência de componentes externos inseguros aumenta a superfície de ataque. Uma boa prática é incluir no processo de desenvolvimento seguro ou pipeline *DevSecOps* a avaliação de segurança do componente externo novo. Há diversos exemplos de possíveis problemas de segurança com componentes de terceiros: carregar uma biblioteca com vulnerabilidades conhecidas (CWE, CVE, etc.), incluir uma biblioteca com recursos extras que envolvem riscos de segurança, reutilizar uma biblioteca que não atenda aos padrões atuais de segurança, utilizar serviço de terceiros e passar a responsabilidade da segurança para ele, errar na configuração da segurança

de uma biblioteca (adotar padrões seguros), incluir biblioteca que envia requisições para o site do criador ou algum parceiro, incluir biblioteca que recebe solicitações de alguma fonte externa, usar componente externo com muitos níveis de inclusão ou dependência "recursiva" e, por último, incluir componentes com interfaces desconhecidas, tais como, por exemplo, comandos de linha, interface web, autenticação própria, interface de depuração, modo desenvolvedor, *backdoor* de acesso extraordinário).

5.4.1.10. Prever alterações futuras em componentes de segurança

Assim como todo software, os componentes de segurança também sofrem mudanças corretivas e evolutivas. A arquitetura de segurança deve ser flexível ao considerar alterações futuras para evolução dos componentes de segurança. As aplicações e os componentes de segurança devem ser projetados com uma visão das mudanças futuras, tais como: atualizações seguras de partes ou do todo, propriedades de segurança que mudam com o tempo, código é atualizado para versões novas, isolamento total ou parcial de funcionalidades comprometidas em ataques, alterações em objetos mantidos em segredo, alterações nas propriedades de segurança de componentes fora de controle (externos) e alterações nos tipos de permissões e de usuários.

5.4.2. Testes de segurança e vulnerabilidades comuns em APIs REST

Vulnerabilidades de APIs [top 2023] são vulnerabilidades de projeto, semânticas ou da lógicas da aplicação e seguem um racional simples [tes 2020]: em geral, a lógica das aplicações é defeituosa de alguma forma e as falhas lógicas são bastante variadas. As falhas lógicas vão desde defeitos simples em trechos de código até vulnerabilidades complexas na interoperação de componentes. Às vezes, elas são óbvias e fáceis de detectar; outras vezes, são muito sutis e não são percebidas em revisões de código ou testes de intrusão. As falhas lógicas dificilmente são eliminadas por meio de programação segura simples, nem detectadas por análise estática de código ou testes de intrusão comuns.

Detectar e prevenir falhas lógicas muitas vezes requer pensamento lateral (fora da caixa). Por outro lado, há recomendações simples para evitar as vulnerabilidades semânticas, em particular aquelas relacionadas a APIs inseguras [Stuttard and Pinto 2008]: validação (sintática e semântica) dos dados de entrada da lógica de negócios, proteção contra falsificação de transações, requisições, solicitações, pedidos, etc., autotestes (automáticos) de integridade do processamento, proteções contra variações de tempo e canais laterais de tempo, imposição de limites ao número de vezes que funções críticas são chamadas, controle do "passo a passo" dos fluxos de trabalho (*workflows*), controles internos contra "maus usos" das aplicações e APIs, controles contra *upload* de arquivos de tipos errados, corrompidos ou maliciosos, projetar aplicações com as boas práticas de projeto seguro.

As vulnerabilidades semânticas em APIs estão relacionadas ao mau uso ou ao abuso de funcionalidades legítimas das aplicações e podem ser causadas por falhas de projeto e não por bugs/defeitos de codificação insegura. Ataques a estas vulnerabilidades geralmente envolvem fraudes na lógica de negócios da aplicação e e podem ser automatizados para abuso em larga escala [Watson and Zaw 2018]. As próximas subseções mostram como explorar algumas das vulnerabilidades de API mais comuns [top 2023] encontradas no OWASP Juice Shop.

5.4.2.1. Descobrindo a API do Juice Shop

O procedimento a seguir descreve como descobrir a API de uma aplicação a partir da análise do código fonte do *frontend* web com o auxílio das ferramentas do desenvolvedor.

1. Abrir a aplicação Juice Shop no navegador com `http://localhost:3000`.
2. Uma vez na tela/página inicial da aplicação Juice Shop, acessar o código fonte do *frontend* da aplicação a partir das ferramentas do desenvolvedor → Sources.
3. Visualizar o arquivo `main.js` (ou algo parecido) e usar o botão *Pretty printing* ou botão **{}** para deixar o código fonte legível.
4. Buscar (menu três pontos verticais → buscar) pela palavra *"path"*.
5. A API da aplicação pode ser facilmente identificada no resultado da busca.
6. Outras buscas possíveis são "API"e "REST". Diversos *endpoints* de API são facilmente identificáveis nesta busca.
7. Identificar a API do *dashboard* de desafios e acessá-la. *Spoiler alert!* O *dashboard* está em `http://localhost:3000/#/score-board`.

5.4.2.2. Quebra de autorização em nível de objeto

De acordo com OWASP API01:2023 (*Broken Object Level Authorization*) [top 2023], APIs tendem a expor *endpoints* para manipulação de objetos internos, criando uma ampla camada de ataque ao controle de nível de acesso. Para evitar problemas, o controle de autorização deve ser verificado em toda função que tenha acesso à fontes de dados que utilizem dados enviados pelo usuário. O procedimento a seguir descreve como um usuário autenticado consegue visualizar sem autorização a cesta de compras de um outro usuário. O objetivo o ataque é é visualizar a cesta de compras de um usuário, qualquer um, sem estar logado como ele.

1. Criar dois usuários com login/senha, `user1@juice/user1` e `2@juice/user2`.
2. Encher a cesta do `user2` com diversos produtos do JuiceShop.
3. Descobrir e explorar a API da cesta de compras.
4. Exploração 1 (exploração do armazenamento local):
   (a) abrir as ferramentas de desenvolvedor.
   (b) Entrar o App JuiceShop logado como `user1`.
   (c) acessar a cesta de compras do `user1`.
   (d) em *DevTools* → *Network*, observar que a API REST da cesta é simples. Por exemplo, `http://localhost:3000/rest/basket/7`. O que é o número 7?
   (e) em *DevTools*, *Application* → *Session Storage* aparecem dois elementos `bid = 7` e `ItemTotal = 0`. será o mesmo número 7?

(f) Conclusão deduzida: bid quer dizer *BasketID*.

(g) Alterar bid para um valor qualquer e atualizar a página. Por exemplo, `bid = 6`!

(h) O que acontece? *Spoiler Alert*! Aparece a cesta de compras do `user2`.

5. Exploração 2 (interceptação da requisição):

   (a) no *OWASP Zap Proxy* do Juice Shop, interceptar e reenviar a requisição do `basket`.

   - Botão direito na requisição do histórico;
   - Na tela de reenvio, substituir o valor de `bid` na requisição por outro valor qualquer até achar um valor válido.

   (b) Esta exploração pode ser automatizada para baixar as cestas de compras de todos os usuários.

   (c) Como poderia ser automatizado e por que este ataque acontece?

5.4.2.3. Autenticação de usuário quebrada

A acordo com OWASP API02:2023 (*Broken User Authentication*) [top 2023], mecanismos de autorização não raramente são implementados incorretamente, permitindo que atacantes comprometam *tokens* de autenticação ou explorem falhas na implementação para assumir identidades temporária ou permanentemente. Isto compromete a habilidade dos sistemas em identificar o usuário/cliente comprometendo a segurança da API. O procedimento a seguir descreve quatro jeitos diferentes de violar o mecanismo de autenticação da aplicação Juice Shop visando obter acesso às credenciais do usuário e logar na aplicação de maneira indevida.

- Exploração 1 (Vazamento das credenciais dos usuários pela API):

  1. Logar na aplicação como `user1@juice`.
  2. no OWASP *ZAP Proxy* do Juice Shop, interceptar e reenviar a requisição de *products/search*.
  3. `GET http://localhost:3000/rest/products/search?q=`.
     - Botão direito na requisição do histórico.
     - Na tela de reenvio, substituir *q=* por *q='))–*. Por que não *q='–* ?
  4. *Spoiler alert*! A requisição *products/search* sofre de SQLi!
  5. Na tela de reenvio, substituir `q=` pelo seguinte *')) UNION SELECT id, email, password, '4', '5', '6', '7', '8', '9' FROM Users–*.
  6. Reenviar a requisição injetada!
  7. Pergunta 1: Como sei que o nome da tabela é *Users*? Deduzindo que na API REST, serviços são iguais a tabelas tabelas.
  8. Pergunta 2: Como sei a quantidade de campos? Por tentativa e erro.

De posse do *dump* da base de dados do usuário, o atacante pode fazer o seguinte descobrir a senha dos usuários.

- Na resposta da requisição, procurar pelo `user1@juice`.
- O *hash* da senha desse usuário está no campo *description*.
- Por exemplo, em `24c9e15e52afc47c225b757e7bee1f9d`.
- No website `https://crackstation.net/`:
  - Digitar o *hash* e apertar o botão *CrackHashes*.
  - Deu um *match* perfeito com md5 e user1 e assim descobre-se o algoritmo de *hash* usado na criptografia da senha!
- Repetindo o processo para o usuário admin:
  - O *hash* é `0192023a7bbd73250516f069df18b500`.
  - O *website* `https://crackstation.net/` diz que a senha é `admin123`.
- Tarefa: descobrir a senha dos outros administradores!

As próximas explorações são mais diretas sobre a interface de login e a API de autenticação. Elas exploram uma vulnerabilidade de injeção de SQL sobre a API ou interface de login.

- Exploração 2 (SQLi sem conhecer usuário algum)
  1. Logar na aplicação com *login* `' or 1=1-` e qualquer senha.
  2. A aplicação vai logar como a primeira entrada na tabela *Users*.
  3. Por que é o Admin?
- Exploração 3 (SQLi com e-mail de administrador).
  1. Logar na aplicação com *login* `admin@juice-sh.op'-` e qualquer senha.
  2. 2a. parte da consulta SQL é comentada e a senha não é verificada.

A próxima exploração faz um ataque de dicionário sobre a senha fraca do administrador.

- Exploração 4 (Ataque de dicionário de senhas fracas)
  1. Logar na aplicação com *login* `admin@juice-sh.op` e senha `admin123`.
  2. Senha *default* descoberta por tentativa e erro ou busca exaustiva com dicionário de senhas fracas.
  3. Faze a busca exaustiva.

5.4.2.4. Autorização quebrada em nível de propriedade de objeto

De acordo com OWASP API3:2023 (*Broken Object Property Level Authorization*) [top 2023], ao permitir que um usuário acesse um objeto por meio de um *endpoint* de API, é importante validar se o usuário tem acesso às propriedades específicas do objeto que estão sendo acessadas. Um *endpoint* de API está vulnerável se: o *endpoint* de API expõe propriedades de um objeto que são consideradas sensíveis e não devem ser lidas pelo usuário (anteriormente denominado: "Exposição Excessiva de Dados") ou o *endpoint* de API permite que um usuário altere, adicione ou exclua o valor de uma propriedade sensível do objeto, à qual o usuário não deveria ter acesso (anteriormente denominado: "Atribuição em Massa"). O procedimento a seguir descreve como usar uma API para acessar mais informação do que é mostrada pela interface gráfica da aplicação.

- Abrir a aplicação Juice Shop com OWASP Zap Proxy:
  - Logar como administrador (usar qualquer dos métodos anteriores).
  - Acessar `http://localhost:3000/#/administration`, aparece uma lista de usuários.
  - clicar em qualquer um dos usuários da lista (ícone de olho).
- No OWASP ZAP:
  - Verificar no histórico de requisições que a API Users foi ativada.
  - Por exemplo, `http://localhost:3000/api/Users/2`, o que significa o 2?
  - Reenviar esta requisição com outros parâmetros.
  - Por exemplo, 3, 5, 8 devolve o json completo do usuário correspondente.
  - Sem parâmetros, devolve o json de todos os usuários.

5.4.2.5. Consumo irrestrito de recursos

De acordo com OWASP API4:2023 (*Unrestricted Resource Consumption*) [top 2023], atender a solicitações da API requer recursos como largura de banda de rede, CPU, memória e armazenamento, podendo até ser pagos por solicitação, como no caso de envio de e-mails/SMS/ligações telefônicas, validação biométrica, por exemplo. Uma API está vulnerável ao uso irrestrito de recursos se pelo menos um dos seguintes limites estiver ausente ou configurado de forma inadequada (por exemplo, muito baixo/alto): tempos limite de execução, memória máxima alocável, número máximo de descritores de arquivo, número máximo de processos, tamanho máximo de arquivo para *upload*, número de operações a serem executadas em uma única solicitação de cliente da API (por exemplo, agrupamento de GraphQL), número de registros por página para retornar em uma única solicitação-resposta, limite de gastos de provedores de serviços de terceiros.

Por exemplo, no Juice Shop, a falta de limites para a quantidade de requisições de login torna possível um ataque de força bruta visando testar em massa muitas opções de senha para um usuário. Este ataque pode ser realizado com o auxílio da ferramenta de *fuzzing* (chamada de *fuzzer*) do OWASP ZAP. O passo a passo é o seguinte: gera uma requisição de login incorreto

para alimentar o *fuzzer*, ativa o *fuzzer* nessa requisição no histórico, adiciona um Fuzzer de Strings com as senhas de teste (para uma quantidade maior de senhas de teste, um *fuzzer* de arquivos pode ser usado) e iniciar o *fuzzer*. O *fuzzer* contorna o *frontend* da aplicação, que fica no mesmo ponto, enquanto o resultado do *brute force* é mostra na janela do *fuzzer*, incluindo qual teste foi bem-sucedido (acertou a senha do usuário).

5.4.2.6. Acesso irrestrito a fluxos de negócios sensíveis

De acordo com OWASP API6:2023 (*Unrestricted Access to Sensitive Business Flows*) [top 2023], ao criar um *endpoint* de API, é importante entender qual fluxo de negócio ele expõe. Alguns fluxos de negócio são mais sensíveis do que outros, no sentido de que um acesso excessivo a eles pode prejudicar o negócio. Exemplos comuns de fluxos de negócio sensíveis e o risco de acesso excessivo associado a eles: fluxo de compra de um produto, fluxo de criação de comentário/postagem, fluxo de fazer uma reserva, fluxo de criação de usuários. Um *endpoint* de API está vulnerável se expõe um fluxo de negócio sensível sem restringir adequadamente o acesso a ele. Por exemplo, no Juice Shop, é possível criar vários usuários e massa porque a API de criação de usuário tem acesso sem autenticação. Com uma ferramenta de desenvolvimento de APIs, como o Postman, é possível contornar o *frontend* e criar vários usuários.

5.4.2.7. Configurações de segurança mal feitas

De acordo com OWASP API8:2023 (*Security Misconfiguration* [top 2023], existem diversas situações que fazem uma API vulnerável: falta de fortalecimento de segurança adequado em qualquer parte da pilha da API; permissões configuradas de forma inadequada nos serviços de nuvem; sistemas estão desatualizados; recursos desnecessários estão habilitados (por exemplo, verbos HTTP, recursos de registro); falta de *Transport Layer Security* (TLS); diretrizes de segurança ou controle de cache não são enviadas aos clientes; política de *Cross-Origin Resource Sharing* (CORS) está ausente ou configurada de forma incorreta; mensagens de erro incluem rastreamentos de pilha ou expõem outras informações sensíveis.

Por exemplo, no Juice Shop, uma configuração insegura do servidor web pode favorecer um ataque de *path traversal*. Na página *About Us*, procurar um link para a URL `http://localhost:3000/ftp/legal.md`. A pasta ftp está aberta e pode ser acessa diretamente pelo caminho `http://localhost:3000/ftp`. Serviços gratuitos de varredura de vulnerabilidades HTTP/HTTPS podem ser usados para detectar falhas de configuração ou configurações inseguras em cabeçalhos HTTP/ HTTPS. Três dos mais conhecidos são o *Mozilla Observatory* (`https://observatory.mozilla.org`), o *Security Headers* (`https://securityheaders.com/`) e o *Qualys SSL Labs – SSL Server Test* (`https://www.ssllabs.com/ssltest/`).

5.4.2.8. Consumo inseguro de APIs

De acordo com OWASP API10:2023 (*Unsafe Consumption of APIs*) [top 2023], desenvolvedores tendem a confiar mais nos dados recebidos de APIs de terceiros ou de *frontends* inseguros do que na entrada do usuário e, portanto, tendem a adotar padrões de segurança mais fracos. Para

comprometer APIs, os atacantes focam em serviços ou códigos de terceiros integrados em vez de tentar comprometer diretamente a API alvo. Na versão, os problemas de injeção tinha sua própria categoria API08:2019 - *Injection*. O procedimento a seguir descreve como explorar uma vulnerabilidade *Cross-Site Scripting* (XSS) refletido no campo de busca da aplicação Juice Shop. É curioso observar que o ataque XSS até poderia ser realizado sobre a API, mas o resultado somente é visível na interface web da aplicação.

- Entrar na aplicação Juice Shop
- Logar na aplicação com usuário e senha (criar usuário, se necessário)
- Fazer uma busca por um nome de fruta (em inglês), O que acontece?
- O campo de busca exibe de volta para o usuário as palavras buscadas!
- O campo de busca aceita tags HTML?
- Testar com: `<h1> orange`.
- O campo de busca aceita javascript?
- Testar com: `<script>alert(1)<\scritp>`.
- A busca aceita DOM?
- Testar com: `<iframe src="javascript:alert('xss')">`.
- O resultado do XSS é visível na interface!!!

5.4.3. Segurança em *JSON Web Tokens*

JSON Web Token (JWT) é um padrão aberto (RFC 7519) que define uma forma compacta e autocontida de transmitir informações de forma segura entre as partes comunicantes como um objeto JSON. Essas informações podem ser verificadas e consideradas confiáveis porque são assinadas digitalmente. JWTs podem ser assinados usando um segredo (com o algoritmo HMAC) ou um par de chaves pública/privada usando RSA ou ECDSA.

Existem diversas maneiras incorretas de usar um JWT que levam a vulnerabilidades exploráveis em ataques reais. A primeira e mais comum é gerar um *token* JWT assinado e não verificá-lo ao recebê-lo. Por exemplo, ao gerar um *token* sem assinatura (`"alg":"none"`) e, por isso, sem capacidade de preservação de integridade nem de autenticidade.

Outra vulnerabilidade é a revelação de segredos (senhas e chaves) embutidos no JWT. Um *token* JWT comum não garante sigilo e não pode ser utilizado para transporte de segredos. Por exemplo, no caso de uso incorreto em que a chave secreta do HMAC é embutida no próprio *token* JWT. Também existe a possibilidade de confusão entre tecnologias criptográficas, quando o desenvolvedor, incorretamente, trata uma chave pública RSA ou ECDSA embutida no *token* como se ela fosse uma chave secreta HMAC e usada como *tag* de autenticação.

O Juice Shop possui um exemplo didático de *token* JWT inseguro. Ao aceitar o login de um usuário legítimo, a aplicação devolve várias informações úteis, entre elas o *token* de

autenticação da sessão do usuário. Esse *token* pode ser copiado (via acesso a resposta da requisição de login) e analisado no website `https://jwt.io`, em que podem ser observadas algumas das vulnerabilidades citadas nesta seção.

5.5. Proteção de aplicações web com WAF

A seção mostra o funcionamento do ModSecurity, um *firewall* de aplicações web (WAF de HTTP), e seu conjunto de regras padrão (*Core Rule Set* – CRS) para detectar, alertar e bloquear ataques baseados na exploração de vulnerabilidades conhecidas. A seção também mostra como um WAF pode falhar ao não ser capaz de detectar vulnerabilidades semânticas relacionadas à lógica de negócios das aplicações.

Existem duas maneiras de incluir um WAF na arquitetura de uma aplicação web. A primeira é embutindo o WAF no mesmo servidor web em que a aplicação está executando. Esta opção tem a vantagem de simplificar a arquitetura geral e a desvantagem de onerar o servidor web da aplicação. Por outro lado, o fortalecimento de segurança pode ser feito de modo específico em cada aplicação, como um *patch* virtual. A segunda opção é separar o WAF em outro servidor web diferente daqueles usados pelas aplicações, em uma arquitetura conhecida como *proxy* reverso. Além da função de segurança, este servidor web pode incluir funções de alta disponibilidade e balanceamento de carga. Neste caso, o fortalecimento de regras é geral a toda a arquitetura, mas ainda assim pode receber o nome de *patch* virtual.

Neste texto, o WAF está implantado em um *proxy* reverso, aquele que, em uma arquitetura cliente-servidor, fica próximo ao servidor e serve de intermediário na comunicação das aplicações web com os clientes (navegadores) na Internet. No exemplo de WAF que se segue, os seguintes elementos de arquitetura são identificáveis:

- Aplicação web: alguma das aplicações vulneráveis (Juice Shop, bWAPP, WebGoat).
- *Proxy* reverso: implantação do servidor web apache2 com o modproxy instalado.
- *Firewall* de aplicação: modsecurity com *Core Rule Set* (CRS) implantado no apache2.
- (Opcional) *Proxy* do navegador: OWASP ZAP ou o Burp *Community* em modo *proxy*.
- Cliente ou *frontend*: navegador web ou cliente de API, como o Postman.

5.5.1. Configuração do WAF ModSecurity no Servidor Web Apache

A listagem 5.15 mostra os comandos instalação do servidor web Apache e o ModSecurity neste servidor em uma máquina Ubuntu. Primeiro, ocorre a instalação do servidor web e o teste das funções básicas do servidor. Em seguida, habilitam-se os módulos de *proxy* do Apache. Finalmente, ocorre a instalação do ModSecurity para Apache. Como boa prática, o Apache é reiniciado depois de cada tarefa e, se a página padrão do Apache aparecer em `http://localhost` após cada tarefa, a implantação está funcionando corretamente. As configurações de segurança *default* do WAF (isto é, as configurações que determinam a rigidez com que as regras são aplicadas) não foram modificadas.

Listagem 5.15. Instalação do ModSecurity no Apache.

```
1 $ sudo apt-get update
```

```
2 $ apt-get install apache2
3 # Para testar, abrir o navegador e acessar http://localhost
4
5 # Para parar|inciar|reiniciar|recarrgar configs|habilitar|desabiliatr o servico
6 $ sudo systemctl stop | start | restart | reload | disable | enable apache2
7
8 # Instalacao e/ou habilitacao dos modulos proxy do apache
9 $ sudo a2enmod proxy proxy_http proxy_balancer lbmethod_byrequests
10
11 # Restart do Apache
12 $ sudo systemctl restart apache2
13
14 #Download e instalacao do modulo ModSecurity Apache
15 $ sudo apt install libapache2-mod-security2
16
17 # Restart do Apache
18 $ sudo systemctl restart apache2
19
20 #verificando se a versao instalada eh maior ou igual a 2.9
21 $apt-cache show libapache2-mod-security2
22
23 $ systemctl restart apache2
```

As configurações tanto do Apache quanto do ModSecurity prosseguem com a edição dos arquivos de configuração. Nas distribuições Ubuntu recentes, o Apache é instalado em `/etc/apache2` e o ModSecurity em `/etc/modsecurity`.

- No servidor web, as configurações são as seguintes:
  - No arquivo `/etc/apache2/envvars`, configurar o local dos logs para `/etc/apache2/log` na variável `APACHE_LOG_DIR=/etc/apache2/log`. Isto vai facilitar a inspeção dos arquivos de log.
  - No arquivo `/etc/apache2/mods-enable/security2.conf`, configurar a carga das configurações e das regras do WAF se o módulo mod_security estiver carregado (as regras estão em `/etc/modsecurity/rules`).
    - `IncludeOptional /etc/modsecurity/*.conf`
    - `Include /etc/modsecurity/rules/*.conf`
  - No arquivo `/etc/apache2/sites-enabled/000-default.conf`, configurar os *hosts* virtuais das aplicações como, por exemplo, ilustrado para o Juice Shop na listagem 5.16.
- No ModSecurity, as configurações são as seguintes:
  - Implantar o *Core Rule Set* (CRS) do OWASP (`https://coreruleset.org/`).
    - Baixar o arquivo de regras de `https://github.com/coreruleset/coreruleset/releases`. Aqui, usa-se a versão 3.2.1 do CRS para ModSecurity 2.9.3.
    - Copiar a pasta de regras (`rules`) para a pasta `/etc/modsecurity/rules`.
    - Copiar o arquivo `crs-setup.conf` para a pasta `/etc/modsecurity`.
    - Copiar o arquivo de configuração *default* ModSecurity para um novo `cp modsecurity.conf-recommended modsecurity.conf`.

- Fazer as seguintes configurações no arquivo `modsecurity.conf`:
  - `SecAuditLog /etc/apache2/log/modsec_audit.log`.
  - Função de segurança do WAF em detecção apenas `SecRuleEngine DetectionOnly` ou bloqueio `SecRuleEngine On`.

Listagem 5.16. Exemplo de arquivo *000-default.conf*.

```
1 ## Juice Shop
2 <VirtualHost *:80>
3     ProxyPreserveHost On
4     ProxyPass         / http://127.0.0.1:3000/
5     ProxyPassReverse / http://127.0.0.1:3000/
6 </VirtualHost>
```

Cada uma das aplicações vulneráveis, na nova configuração com WAF, foi novamente varrida pelo *scanner* DAST. Ao confrontar o *scanner* DAST contra o WAF sobre as aplicações vulneráveis, percebeu-se uma redução na quantidade total de alertas de vulnerabilidades das aplicações. Por outro lado, o *scanner* DAST alertou sobre possíveis vulnerabilidades no WAF ou no servidor web Apache usado como *proxy*, que podem ser falsos positivos. Apesar deste ser um resultado positivo, este não é o maior benefício do WAF. As próximas seções mostram que o maior benefício de um WAF está na proteção contra os ataques específicos de aplicações web, como por exemplo, aqueles do OWASP TOP 10 [top 2021].

5.5.2. Bloqueio de XSS, SQLi e RCE com ModSecurity

Todas as ações de detecção ou bloqueio realizadas pelo ModSecurity ficam registradas no arquivo `/etc/apache2/log/modsec_audit.log`. A inspeção deste arquivo é bastante útil para entender como funciona a detecção e bloqueio dos ataques. O ModSecurity é capaz de detectar e bloquear diversos ataques comuns que exploram vulnerabilidades simples as aplicações web, tais como os ataques de injeção de *script* (XSS), injeção de SQL (SQLi) e de execução remota de comandos do sistema operacional (RCE). A implantação padrão do ModSecurity usa as configurações de segurança mais leves, de rigor menor. Diversos testes de vulnerabilidades realizados ao longo deste texto podem ser facilmente detectados (ou bloqueados) pelo WAF com a configuração padrão, a saber (a lista a seguir não é exaustiva):

- Na aplicação Juice Shop, seguintes vulnerabilidades:
  - Injeção de SQL (`' or 1 = 1 #`) no campo de login.
  - XSS simples (`<script>alert(1)</script>`) no campo de busca.
- Na aplicação WebGoat, as seguintes lições:
  - *Injection Flaws* → *String SQL Injection*: `' or 'a'='a' -`.
  - *Injection Flaws* → *Numeric SQL Injection*: `101 or 1=1`.
  - *Cross-Site Scripting (XSS)* → *Reflected XSS Attacks*: `<script>alert(1)</script>`.
  - *Cross-Site Scripting (XSS)* → *Stored XSS Attacks*: `<script>alert(1)</script>`.

  - *Cross-Site Scripting (XSS) → Reflected XSS Attacks*: `<a onmouseover="alert(1)" href="#">read this!</a>`
  - *Injection Flaws → Command Injection*: `" & ls -l ."`.
  - *Ajax Security → Dangerous Use of Eval*: `<script>alert(1)</script>`.

- Na aplicação bWAPP, os seguintes *bugs*:
  - *SQL Injection (Search/GET)*: `' or 1=1 #`.
  - *SQL Injection (Login Form)*: `' or 1=1 #`.
  - *XSS - Reflected (GET)*: `<script>alert(1)</script>` separado em duas partes.
  - *XSS - Reflected (JSON)*: `"}]}'; </script><script>alert(1);</script>`.
  - *PHP Eval function*: `http://localhost/php_eval.php?eval=echo shell_exec ("ls -l");`.
  - *Directory Traversal - Files*: `http://localhost/directory_traversal_1.php?page=../../../../../../../../../etc/passwd`.

5.5.3. Falsos negativos do ModSecurity

O ModSecurity, assim como qualquer outra ferramenta de segurança que tenta classificar um *payload* recebido externamente, sem executá-lo, em benigno ou malicioso, pode cometer enganos e não alertar sobre problemas reais. Essas omissões são chamadas de falsos negativos. De modo geral, quanto mais rica a semântica do ataque, mais difícil a sua detecção pelas ferramentas automáticas.

Diversos testes de vulnerabilidades realizados ao longo teste texto NÃO são detectados nem bloqueados pelo ModSecurity na configuração padrão de instalação, a saber (a lista a seguir não é exaustiva):

- Na aplicação Juice Shop, seguintes vulnerabilidades:
  - Todas as vulnerabilidades de APIs derivadas de vulnerabilidades semânticas ou de falhas de *design*.
  - DOM XSS (`<iframe src="javascript:alert('xss')">`) no campo de busca.
  - *Directory Traversal - Directories*: `http://localhost/ftp`.
- Na aplicação WebGoat, as seguintes lições:
  - *Injection Flaws → Command Injection*: `" & dir ."`.
  - *Ajax Security → Dangerous Use of Eval*: `123'); alert(1);('`.
  - *Parameter Tampering → Bypass Client Side JavaScript Validation*.

  - *Parameter Tampering → Exploit Hidden Fields*

- Na aplicação bWAPP, os seguintes *bugs*:
  - *HTML Injection - Reflected (POST)*: `<h1><a href="http://www.google.com">Link!</a></h1>`.
  - *HTML Injection - Reflected (GET)*: `<h1><a href="http://www.google.com">Link!</a></h1>`.
  - *HTML Injection - Stored (blog)*: `<h1><a href="http://www.google.com">Link!</a></h1>`.
  - *Directory Traversal - Directories*: `http://localhost/document`.

É possível avaliar se estes falsos positivos ainda ocorrem com outros níveis de rigor (paranoia) do ModSecurity. Em uma instalação nativa do *Core Rule Set*, o nível de paranoia é definido pela variável `tx.paranoia_level` no arquivo `crs-setup.conf`. Existem 4 níveis de paranoia. O nível 1 oferece segurança básica com a mínima necessidade de ajustar falsos positivos. O nível 2 é adequado quando dados de usuários reais estão envolvidos e pode apresentar falsos positivos. O nível 3 oferece segurança de nível bancário online e apresenta muitos falsos positivos, que devem ser tratados com regras de exceção. Finalmente, o nível 4 possui as regras mais restritivas e possivelmente com mais falsos positivos que os outros níveis.

5.6. Comparação de ferramentas, falsos positivos e falsos negativos

Esta seção compara as ferramentas SAST, SCA, DAST e WAF, ilustrando as falhas das ferramentas SAST e SCA em relação às DAST e WAF e vice-versa. Considera-se nesta comparação a implantação *out-of-the-box* das ferramentas.

No geral, ferramentas de análise estática de código (seja código fonte ou binário), visando segurança, tendem a ser conservadoras e pessimistas, emitindo alertas mesmo quando não há muita confiança nem certeza na conclusão da análise, visando prevenir o máximo possível de ameaças. Este comportamento produz muitos falsos positivos (alarmes falsos). As ferramentas SAST usadas pelos desenvolvedores em esquadrões ágeis devem ser configuradas com a sensibilidade mínima a fim de que emitam alertas apenas quando houver bastante confiança (certeza?) de que existe de fato uma vulnerabilidade explorável. Este decisão de configuração diminui a quantidade de falsos positivos, poupando tempo do *Security Champion* que iria para analises desnecessárias. Por outro lado,a ferramenta de cobertura minimalista tem menor capacidade de detecção de problemas, possivelmente, aumentando os falsos negativos.

Geralmente, a cobertura das ferramentas de análise de código (fonte ou binário) é limitada pela incapacidade destas ferramentas de reconhecem todas as instâncias de vulnerabilidades documentadas e suas variações em situações específicas. Além disso, podem haver limitações de ferramentas em percorrer todos os caminhos e fluxos de controle em códigos fonte, tendo bastante dificuldade em identificar vulnerabilidades em trechos de código ou estruturas de dados complexas, resultando em pontos cegos onde as ferramentas não são capazes de enxergar vulnerabilidades, que são omitidas dos alertas. A omissão de um alerta sobre uma vulnerabilidade verdadeira em um falso negativo da ferramenta.

O desempenho das ferramentas SAST pode ser analisado com segue. A ferramenta SonarQube obteve um desempenho relativamente satisfatório em relação ao WebGoat (poucos

falsos positivos e muitos falsos negativos) e modesto em relação ao Juice Shop (poucos positivos verdadeiros e muitos falsos negativos), tendo decepcionado para o bWAPP e a Gruyere (em ambos, majoritariamente falsos negativos). Isto se deve ao fato de haver uma diversidade grande de tecnologias de software novas e, em muitos casos, a versão do *scanner* disponível gratuitamente está bastante desatualizada em relação a essas novas tecnologias.

A ferramenta HoruSec apresentou desempenho bastante interessante, com relatórios longos recheados de alertas para Juice Shop, WebGoat e bWAPP. Mesmo assim, muitos alertas são falsos positivos. Uma vez que HoruSec possui *scanners* mais atualizados que o SonarQube, os alertas obtidos cobrem tecnologias de software mais modernas, como, por exemplo, TypeScript e JWT, além de uma capacidade melhor de detecção de segredos embutidos no código fonte.

O desempenho das ferramentas SCA foi o seguinte. A ferramenta Dependency-Check apresentou uma quantidade de alertas bastante menor que a ferramenta Trivy porque esta última foi utilizada sobre imagens Docker e, por isso, teve acesso a mais dados para análise, não apenas as bibliotecas de terceiros, como aconteceu com o Dependency-Check. Mais artefatos analisados levam a uma superfície de ataque maior e a mais descobertas de vulnerabildiades. As duas ferramentas se complementam quando uma é usada em testes de integração e construção iniciais e a outra é usada mais a frente na esteira de desenvolvimento, quando já existe uma imagem Docker para análise. Uma vez a análise com ferramenta SCA cobre dependências externas e componentes de terceiros, as vulnerabilidades encontradas são diferentes daquelas encontradas pelas ferramentas SAST, que analisam o código fonte. Por isso, a interseção entre os resultados das análises SAST e SCA é mínima. As SCAs apresentam menos falsos positivos que as SASTs porque seus resultados tendem a ser associados a CVEs, considerando-se as mesmas aplicações em análise.

O desempenho das ferramentas DAST pode ser analisa do seguinte modo. As DASTs, quando utilizadas como *scanners* dinâmicos de vulnerabilidades (por exemplo, o ZAP em linha de comando), apresentam menos falsos positivos que SASTs e SCAs porque seus resultados são, no geral, explorações testadas sobre a aplicação em funcionamento. O ZAP em modo *scanner* apresentou bons resultados mesmo para as aplicações ignoradas pelas outras ferramentas (bWAPP e Gruyere). Isto se deve ao fato de todas as aplicações vulneráveis possuírem configurações inseguras de HTTP e não usarem HTTPS. Já, quando usadas em testes manuais, elas auxiliam as explorações de vulnerabilidades reais. Porém, nestes casos, a ferramenta DAST pode possuir uma cobertura menor da aplicação devido a dificuldade em percorrer todos os caminhos de dados e fluxos de execução e controle da aplicação em análise.

Finalmente, como último desafio de análise, o desempenho da ferramenta WAF foi o seguinte. Ao confrontar o *scanner* DAST contra o WAF ModSecurity de configuração padrão sobre as aplicações vulneráveis, percebeu-se uma pequena redução na quantidade total de alertas de vulnerabilidades das aplicações. Por outro lado, o *scanner* DAST alertou sobre uma possível vulnerabilidade no WAF, que pode ser um falso positivo.

Na análise do WAF contra testes de segurança manuais, como os realizados ao longo deste texto, o *firewall* de aplicação web ModSecurity de configuração padrão se mostrou eficiente em detectar vulnerabilidades técnicas simples, como aquelas do OWASP Top 10, mas não foi capaz de detectar aquelas vulnerabilidades de semântica complexa, como as vulnerabilidades de API, tipicamente descobertas em testes de intrusão. Por outro lado, o WAF pode ser personalizado pela adição de regras exclusivas para o contexto de uso, assim como também o aumento de rigor no *Paranoia Level*, o que deve permitir resultados melhores que estes

obtidos com as configurações *out-of-the-box* do ModSecurity e do *Core rule Set* do OWASP.

5.7. Considerações finais

Este texto apresentou o conceitos de desenvolvimento de software seguro ágil e aspectos de cultura *DevSecOps* relevantes para a segurança de aplicações quando promovida por *Security Champions* dentro de esquadrões de desenvolvimento de software.

Quatro tipos de ferramentas de segurança de aplicações (SAST, SCA, DAST e WAF) foram analisadas e comparadas entre si visando esclarecer sobre a melhor maneira de usá-las em conjunto, de forma complementar, em uma esteira de desenvolvimento de software seguro. O desempenho das ferramentas foi analisado por meio de testes e avaliações práticas de segurança contemplando suas capacidades de detecção de vulnerabilidades em aplicações sabidamente vulneráveis.

A análise comparativa de ferramentas de segurança ao longo de um processo de desenvolvimento de software e avaliação sistemática do desempenho de ferramentas SAST não são novidades e já podem até ser encontradas sobre nichos específicos [Braga and Dahab 2015a, Braga et al. 2017, Braga et al. 2019]. Por outro lado, a cultura de segurança ágil e as esteiras *DevSecOps* ainda não são comuns no desenvolvimento de software e sua disseminação ampla é um desafio para a comunidade de software seguro.

A diminuição de falsos positivos e falsos negativos pode contribuir significativamente para o aumento da confiança e, por conseguinte, maior utilização destas ferramentas. Estudos recentes [Rodrigues et al. 2020a, Rodrigues et al. 2020b, Rodrigues et al. 2023] mostram que a utilização de técnicas de aprendizado de máquina podem melhorar muito o desempenho das ferramentas SAST e que a próxima geração destas ferramentas, ao incorporar as técnicas de análise avançadas, será muito superior às ferramentas disponíveis de forma gratuita atualmente. Esta visão otimista pode ser compartilhada com as ferramentas SCA, DAST e WAF.

Agradecimentos

Este trabalho foi realizado pelo grupo de segurança de aplicações do CPQD, dentro do programa ***Security Champions* de Desenvolvimento de Software Seguro** apoiado pela instituição.

Referências

[iee 2014] (2014). Avoiding the top ten security flaws. *URL: https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/Top-10-Flaws.pdf*.

[saf 2019] (2019). Software security takes a champion – a short guide on building and sustaining a successful security champions program. *URL: http://safecode.org/wp-content/uploads/2019/02/Security-Champions-2019-.pdf*.

[tes 2020] (2020). Owasp web security testing guide v4.2. *URL: https://owasp.org/www-project-web-security-testing-guide*.

[top 2021] (2021). Owasp top 10 – 2021. *URL: https://owasp.org/Top10/*.

[dev 2021] (2021). Six pillars of devsecops series. *URL: https://cloudsecurityalliance.org/blog/2021/09/09/six-pillars-of-devsecops-series/*.

[top 2023] (2023). Owasp api security top 10. *URL: https://owasp.org/API-Security/*.

[Braga and Dahab 2015a] Braga, A. and Dahab, R. (2015a). A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software. In *XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, pages 30–43, Florianópolis, SC, Brazil.

[Braga and Dahab 2015b] Braga, A. and Dahab, R. (2015b). Introdução à Criptografia para Programadores: Evitando Maus Usos da Criptografia em Sistemas de Software. In *Caderno de minicursos do XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2015*, pages 1–50. Sociedade Brasileira de Computação.

[Braga and Dahab 2016] Braga, A. and Dahab, R. (2016). Mining Cryptography Misuse in Online Forums. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 143–150.

[Braga and Dahab 2017] Braga, A. and Dahab, R. (2017). A Longitudinal and Retrospective Study on How Developers Misuse Cryptography in Online Communities. In *XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg'17)*, Brasília, DF, Brazil.

[Braga and Dahab 2018] Braga, A. and Dahab, R. (2018). Criptografia assimétrica para programadores - evitando outros maus usos da criptografia em sistemas de software. In *Caderno de minicursos do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais — SBSeg 2018*, pages 1–50. Sociedade Brasileira de Computação.

[Braga and Dahab 2019] Braga, A. and Dahab, R. (2019). Introdução à criptografia para administradores de sistemas com tls, openssl e apache mod_ssl. In *Minicursos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2019)*. Sociedade Brasileira de Computação.

[Braga et al. 2017] Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., and Vieira, M. (2017). Practical Evaluation of Static Code Analysis Tools for Cryptography: Benchmarking Method and Case Study. In *The 28th IEEE International Symposium on Software Reliability Engineering (ISSRE)*.

[Braga et al. 2019] Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., and Vieira, M. (2019). Understanding how to use static analysis tools for detecting cryptography misuse in software. *IEEE Transactions on Reliability*, 68(4):1384–1403.

[Braga et al. 2012] Braga, A., do Nascimento, E. N., da Palma, L. R., and Rosa, R. P. (2012). Introdução à segurança de dispositivos móveis modernos–um estudo de caso em android. *Sociedade Brasileira de Computação*.

[da Silva et al. 2019] da Silva, J., Braga, A., Rubira, C., and Dahab, R. (2019). An approach for adaptive security of cloud applications within the atmosphere platform. In *Anais do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 397–402. SBC.

[Fisher 2022] Fisher, D. (2022). *Application Security Program Handbook—A Guide for Software Engineers and Team Leaders*. Manning Publications Co.

[Institute 2023] Institute, P. M. (2023). *Agile Practice Guide*. Project Management Institute, Newton Square, PA.

[McGraw 2004] McGraw, G. (2004). Software security. *IEEE Security and Privacy*, 2(02):80–83.

[Miller 2022] Miller, N. (2022). Security culture 1.0. *URL: https://owasp.org/www-project-security-culture*.

[Pressman 2018] Pressman, R. S. (c2018.). *Software engineering :*. McGraw-Hill,, Chennai :, 7th ed. edition. Includes index.

[Rodrigues et al. 2020a] Rodrigues, G., Braga, A., and Dahab, R. (2020a). A machine learning approach to detect misuse of cryptographic apis in source code. In *Anais do XX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 1–14, Porto Alegre, RS, Brasil. SBC.

[Rodrigues et al. 2020b] Rodrigues, G. E. d. P., Braga, A. M., and Dahab, R. (2020b). Using graph embeddings and machine learning to detect cryptography misuse in source code. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1059–1066.

[Rodrigues et al. 2023] Rodrigues, G. E. d. P., Braga, A. M., and Dahab, R. (2023). Detecting cryptography misuses with machine learning: Graph embeddings, transfer learning and data augmentation in source code related tasks. *IEEE Transactions on Reliability*, pages 1–12.

[Stuttard and Pinto 2008] Stuttard, D. and Pinto, M. (2008). *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley.

[Watson and Zaw 2018] Watson, C. and Zaw, T. (2018). *OWASP Automated Threat Handbook Web Applications*. OWASP Foundation.