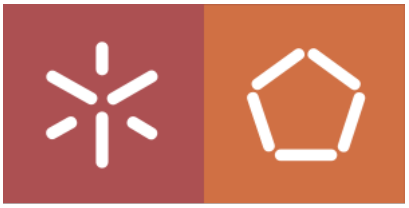

Universidade do Minho
Escola de Engenharia

Engenharia de Segurança

CMD-SOAP

Mariana Fernandes A81728
João Costeira A78073
Paulo Mendes A78203

Conteúdo

1 Introdução

Geralmente é mais barato construir software seguro do que corrigir problemas de segurança após a entrega do mesmo como um produto final ou pacote completo, sem falar nos custos que podem estar associados a uma falha de segurança [1]. O objetivo da segurança em aplicações é manter a confidencialidade, integridade e disponibilidade dos recursos de informação a fim de permitir que as operações de negócios sejam bem sucedidas. Esse objetivo é alcançado através da implementação de mecanismos de segurança.
O objetivo deste trabalho é desenvolver em *NodeJS*, uma aplicação comando linha (CLI) que permita testar as operações do serviço SCMD (Signature CMD), fazendo *reverse engineer* da aplicação CMD-SOAP. Todo o processo deverá seguir padrões de desenvolvimento seguro de aplicações.

2 Adaptação do Código

Na conversão do *software* CMD-SOAP de *Python* para *NodeJS* foram efetuadas algumas alterações devido à adaptação para as funções disponíveis no *NodeJS* e dos *packages* disponíveis.
Com o objetivo de ter uma simples interface via linha de comandos, fez-se uso do *package* `prompt`, que lida com o *input* do utilizador.
Os pedidos ao serviço da Chave Móvel Digital foram feitos usando o *package* `strong-soap`.
O *package* `fs` (*file system*) permite ler e escrever em ficheiros, pelo que foi importado para possibilitar a leitura do ficheiro a assinar.
Para aceder a funções criptográficas de *hash* e de verificação de assinaturas escolheram-se os *packages* `crypto` e `node-forge`.

3 Técnicas de Desenvolvimento Seguro

Construir software seguro exige o conhecimento básico dos princípios de segurança. Estar familiarizado com o OWASP Top 10 e os tipos de *weaknesses* mais comuns é imprescindível, caso contrário, é muito mais provável que se insira uma **bug** ou vulnerabilidade no projeto acidentalmente.

Como lecionado nesta Unidade Curricular, existem certas operações que devem despertar cuidados redobrados pelos impactos que podem ter. Assim, devemos ter em atenção ao comportamento que o *software* deve ter nas seguintes situações, caso surjam:

- Leitura e escrita em *buffers*.

- Operações que possam esgotar o intervalo de dados de uma variável, por exemplo, no caso dos *overflows* e *underflows*.
- *Input* do utilizador.
- Autenticação.
- Uso correto da criptografia.

Existem muitas mais fontes de fraquezas na segurança de uma aplicação, no entanto, no nosso caso em particular, o foco estará mais no *input* do utilizador, por ser uma aplicação CLI, e o uso correto da criptografia, pois é na mesma que se baseiam as assinaturas digitais. De salientar que garantir que o programa tem um comportamento esperado e adequado perante situações inesperadas é igualmente crítico, portanto devem estar implementadas boas práticas no que toca à forma como se lida com os erros.

Para garantir que, de facto, as verificações efetuadas são suficientes e sustentadas, pode-se consultar as *checklists* disponibilizadas pelo *OWASP* no *pdf* 'Melhores Práticas de Programação Segura OWASP - Guia de Referência Rápida' [1].
Como é impossível garantir que não exista erro humano, mesmo que os programadores sejam experientes, é importante utilizar, durante o processo de desenvolvimento, *software* que detete vulnerabilidades. Para este trabalho, optamos por tirar proveito da ferramenta *HCL AppScan CodeSweep*, uma extensão para o *Visual Studio Code*, que faz análise estática do código (*Static Application Security Testing*).
Experimentou-se também usar a extensão *CVE for NodeJS* que mostra alertas de segurança para dependências de projetos *NodeJS* com vulnerabilidades, mas nada foi detetado.

3.1 HCL AppScan CodeSweep

Esta extensão para o *Visual Studio Code* disponibiliza uma lista de possíveis vulnerabilidades presentes no código, atualizando sempre que se grava o documento.
Como podemos constatar pela figura abaixo, o *CodeSweep* encontrou um caso onde se imprime um resultado diretamente para a consola. Por si só, isto não representa uma vulnerabilidade, no entanto, é má prática fazê-lo, pois pode vir a expor dados involuntariamente.

Figura 1: Resultado da análise do HCL AppScan CodeSweep

Durante o desenvolvimento, foram detetadas mais instâncias da mesma categoria, sendo corrigidas em seguida. Assim, evitou-se fazer alterações posteriores ao desenvolvimento, que, num cenário real, podem custar muito tempo e dinheiro.

4 Indicadores de Qualidade - SonarQube

Para produzir alguns indicadores de qualidade, assim como efetuar mais uma análise de segurança ao código, utilizou-se a ferramenta SonarQube, já explorada pelo grupo no trabalho anterior.
Como podemos observar na figura abaixo, numa primeira análise não foram encontradas vulnerabilidades, apenas *Security Hotspots* - considerados possíveis vulnerabilidades- , a serem analisados, e *Code Smells* - más práticas de codificação, que podem tornar a aplicação menos segura e mais difícil de manter.

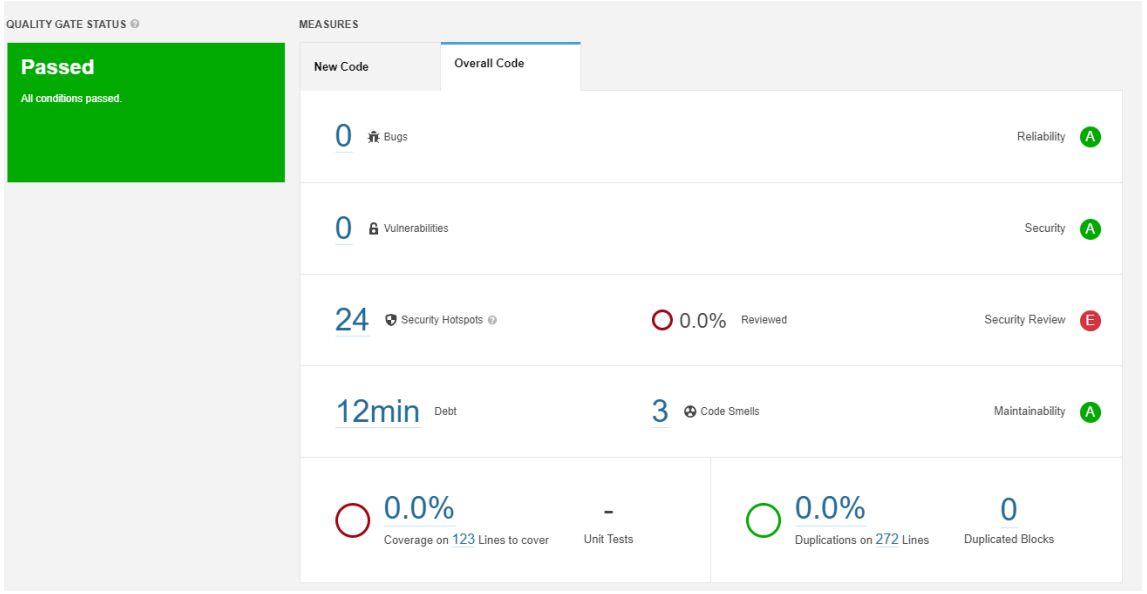

Figura 2: Primeira Análise do SonarQube

Na imagem seguinte, temos o primeiro *code smell*, onde, por distração, faltou referir a *keyword* `var` ou `let` ao declarar a variável. Isto faz com que, acidentalmente, seja criada uma variável global. Basta adicionar o `let` antes da declaração para resolver este *code smell*.

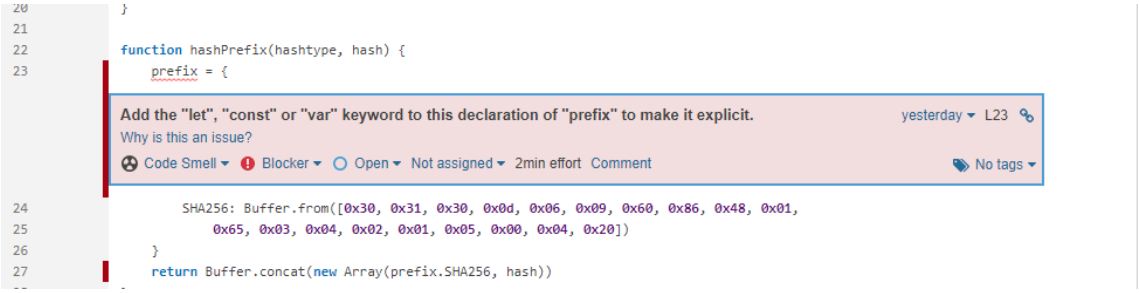

Figura 3: *Code Smell* 1

Depois, temos dois casos seguidos onde uma variável que foi declarada num *scope* exterior, é novamente declarada. Este comportamento é má prática pois torna o código mais difícil de ler, e por consequência, mais difícil de manter. Como `err` e `res` são comummente utilizados ao lidar com promessas, é compreensível o porquê da sua repetição, ainda assim, a situação foi mitigada alterando o nome das variáveis.

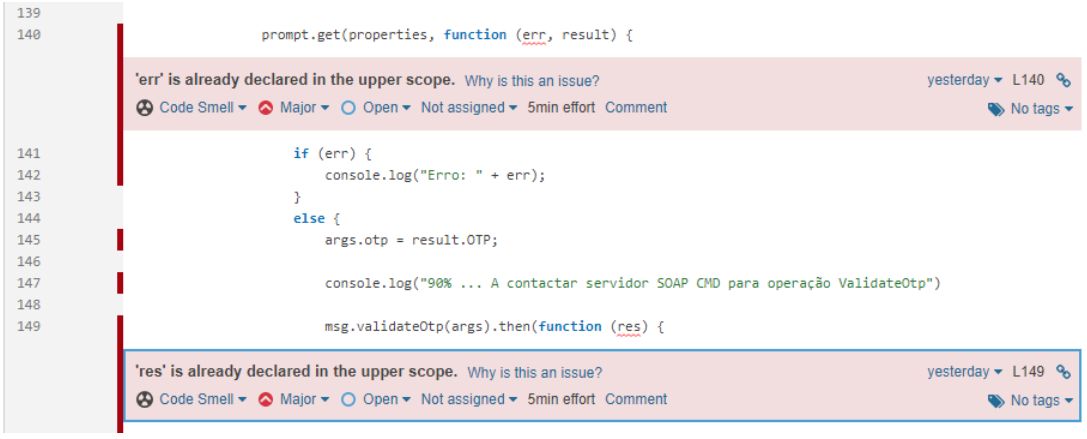

Figura 4: *Code Smell* 2 e 3

Após estas correções, foi feita uma nova análise. Entretanto, como podemos ver, foram introduzidos 4 novos *security hotspots*. Poderá dever-se à mitigação dos *code smells* e alguma alteração entretanto feita durante o desenvolvimento. Podemos notar que o *SonarQube* não apresenta o *code coverage* corretamente, porque requer que o relatório de *coverage* seja fornecido por outro *software*.

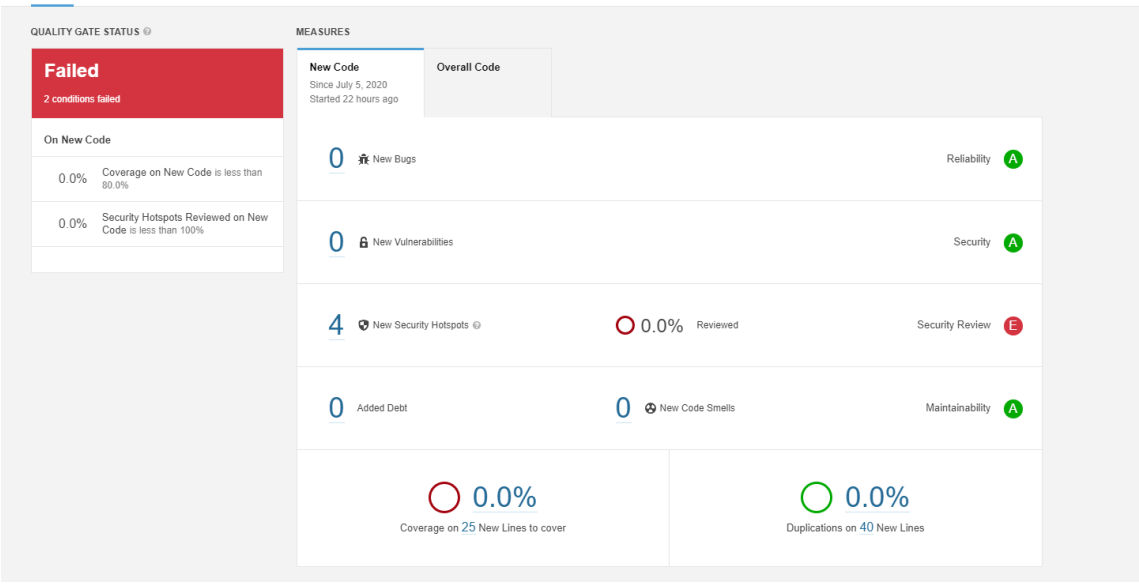

Figura 5: Segunda Análise do SonarQube

Quatro dos *security hotspots* detetados são referentes ao uso de funções de *hash*. Segundo a documentação, esta regra apanha o código onde a *hash* é iniciada, mas não implica que a utilização da mesma não seja segura. Serve como alerta para que não se usem funções criptográficas fracas ou implementadas pelos próprios programadores. No caso do CMD-SOAP, a utilização do `SHA-256` é indispensável, pelo que podemos categorizar estes avisos como

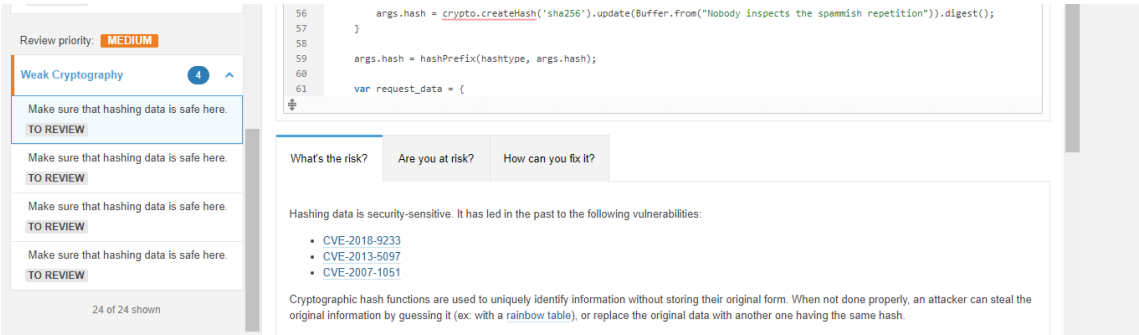

Figura 6: *Security Hotspots* - Criptografia

Figura 7: *Security Hotspots* - Criptografia - Assinalar como seguro

Nesta figura temos a representação de um caso onde o *input* não foi sanitizado. Regra geral, não o fazer pode provocar danos graves nas aplicações, mas como estamos a comunicar com um serviço externo que efetua sanitizações e devolve respostas mesmo que receba valores que não espera, o impacto não é tão crítico.

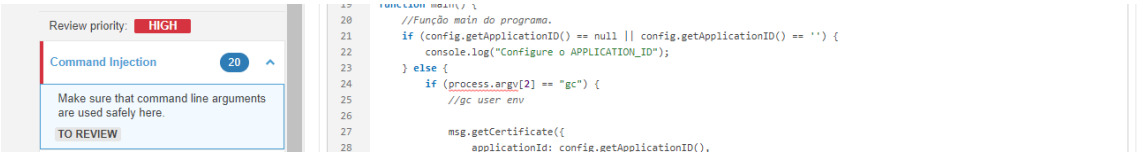

Figura 8: *Security Hotspots* - Sanitização de *Input*

Finalmente, podemos ver que o nosso código passa todas as métricas do SonarQube, sendo que não foi possível testar corretamente a *coverage* e que, apesar de tentarmos alterar a sensibilidade do CPD (*Copy Paste Detector*), não foi detetado código duplicado.

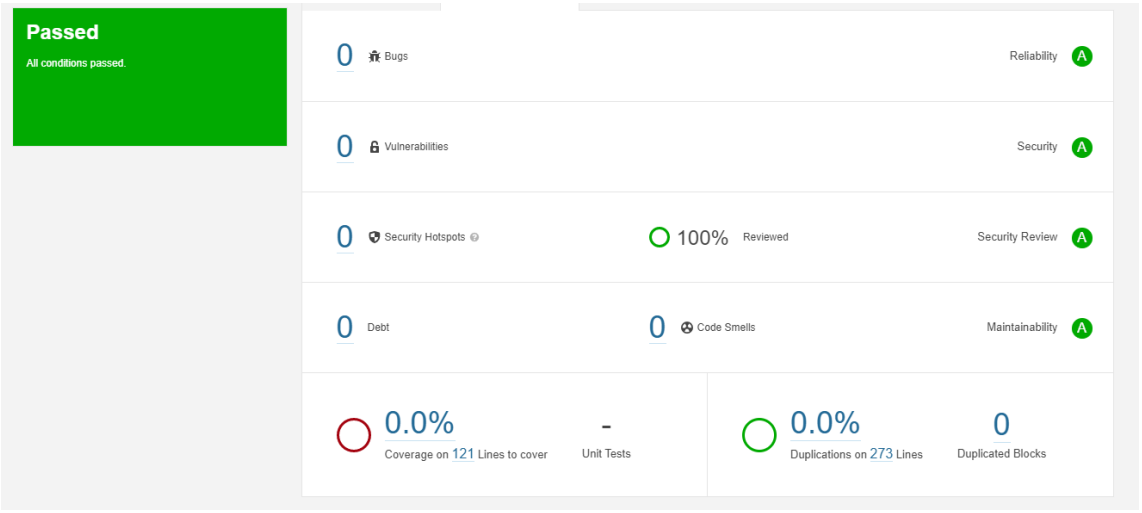

Figura 9: Análise Final do SonarQube

5 Demonstração

Para instalar a aplicação basta correr `npm -i`, que automaticamente instala as dependências necessárias.
Foram disponibilizadas as seguintes funcionalidades:

- gc - Vai buscar o certificado ao servidor
- ms - Assina um documento
- mms - Assina vários documentos
- otp - Autoriza a assinatura, para que fique válida.
- test - Assina um ficheiro e valida a assinatura

Podemos correr o teste com o comando:

```
node . test 'nomedoficheiro' '+351 numerotelemovel' pin
```

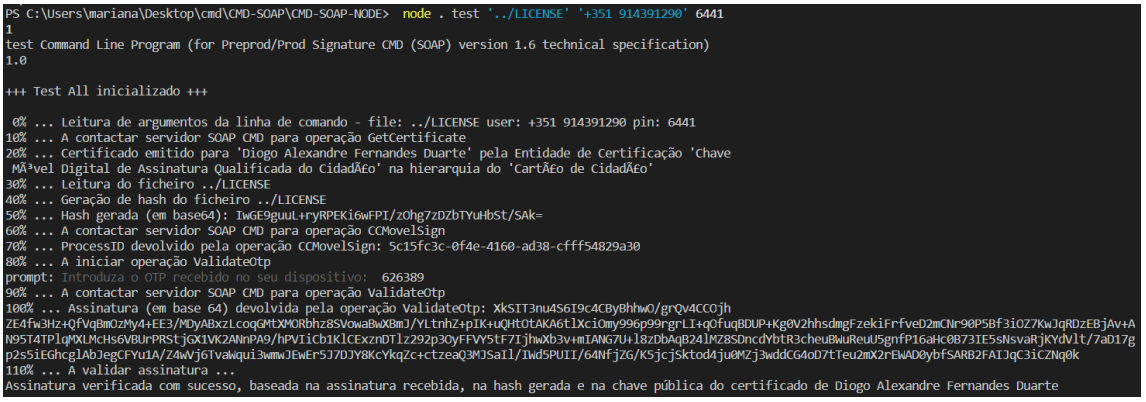

Figura 10: Resultado da função de teste

6 Conclusão e Trabalho Futuro

Futuramente poderíamos adaptar o código para fazer uso de apenas um *package* criptográfico. Como se sabe, os componentes externos devem ser reduzidos quando possível, para ser mais fácil mantermo-nos a par das suas alterações e vulnerabilidades.

Apesar do uso de várias ferramentas de análise de vulnerabilidades, foram detetadas poucas ocorrências. Isto pode ser justificado pelos cuidados que o grupo teve a desenvolver a aplicação, mas também pela qualidade das ferramentas de análise ou até pelo tipo de análise feita, pois a análise estática (SAST) não é capaz de detetar todos os tipos de vulnerabilidades. Tendo isto em conta, seria apropriado fazer uma análise com ferramentas de maior qualidade, que infelizmente são pagas, e fazer testes de penetração à aplicação, caso a mesma fosse disponibilizada ao público.

Em suma, este trabalho sublinha a importância da segurança como parte integrante do processo de desenvolvimento de *software* e a necessidade de sensibilizar as empresas e os programadores para esse mesmo facto.

7 Bibliografia

1. https://owasp.org/www-pdf-archive/OWASP_SCP_v1.3_pt-PT.pdf
2. https://marketplace.visualstudio.com/items?itemName=HCLTechnologies .hclapps-cancodesweep
3. https://www.sonarqube.org/
4. https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/