Universidade do Minho

Mestrado Integrado em Engenharia Informática

Engenharia de Segurança

Projeto 3 - Teste às operações do serviço SCMD

Grupo 2
Paulo Gameiro - A72067
Pedro Rodrigues - PG41092
Rafaela Soares - A79034

Braga, Portugal
6 de Julho de 2020

Conteúdo

1 Introdução

No início do segundo semestre foi realizado um projeto que consistiu num conjunto de orientações para o desenvolvimento seguro de *software* - desde a fase inicial de conceptualização até à utilização. Estas orientações incorporam processos de desenvolvimento dinâmico e cobrem algumas preocupações e abordagens sobre aplicativos *web* e *cloud services*.

Nesse âmbito, foi estudada a publicação *Fundamental Practices for Secure Software Development* da organização SAFECode publicada a março de 2018, sendo que existem publicações mais recentes.

Posteriormente, em meados do segundo semestre foi realizado outro projeto, no qual foram analisadas técnicas e ferramentas de *Abstract Interpretation*. Este método trata-se de um método formal matemático baseado em técnicas formais de verificação que proporcionam aos programadores abstrair a semântica do *software*.

A verificação de programas consiste na melhoria da semântica do programa que satisfaz a sua especificação. A interpretação abstracta formaliza a ideia de que isto é possível com um determinado nível de abstracção, onde os detalhes envolvendo a semântica e a especificação são ignorados. Um exemplo da abstracção de semântica é a lógica de Hoare, enquanto que os exemplos de abstracção de especificação são o invariante, correcção parcial e total.

Por fim, como último projeto, é esperado que se aplique os conceitos vistos e analisados nos projetos anteriores numa aplicação comando linha (CLI), em linguagem RUST, que permita testar as operações do serviço SCMD *(Signature CMD)*, fazendo *reverse engineer* da aplicação CMD-SOAP.

2 *Software* desenvolvido

Como ponto de partida, foi fornecida a aplicação **CMD-SOAP**, desenvolvida na linguagem Python, que permite testar as operações do serviço SCMD *(Signature CMD)*, através de uma Chave Móvel Digital (CMA). Após o estudo desta aplicação e de uma breve introdução à programação em RUST, iniciou-se o desenvolvimento de uma proposta desta mesma aplicação, na linguagem RUST.

2.1 *Libraries*

Numa primeira fase, era fulcral encontrar uma *library* que implementasse o protocolo SOAP (*Simple Object Access Protocol*) em RUST. Para tal, em primeira instância, recorreu-se à *library* soap-rs, disponibilizada **na conta github raventid**. Após o teste desta *library*, contactou-se o programador que a desenvolveu, uma vez que esta apresentava erros nos testes realizados. Como este informou que a *library* ainda não estava pronta para ser utilizada numa aplicação real, tal teve de ser descartada.

Para além disso, recorreu-se também à *library* zeep, disponibilizada **na conta github mibes404**, contudo esta também acabou por ser descartada, assim como a *library* savon. Como alternativa, utilizou-se a *library* hyper.

2.2 Funções relativas às operações do serviço SCMD

- **get_certificate** - testa o comando SOAP GetCertificate do SCMD
- **cc_movel_sign** - testa o comando SOAP CCMovelSign do SCMD
- **cc_movel_multiple_sign** - testa o comando SOAP CCMovelMultipleSign do SCMD
- **validate_otp** - testa o comando SOAP ValidateOtp do SCMD

2.3 Estrutura do código

O projeto foi feito através do Cargo, gerente de pacotes Rust. Como este "procede ao *download* das dependências dos pacotes necessários do programa RUST em questão, compila os pacotes, faz pacotes distribuíveis e carrega-os para crates.io", considerou-se uma opção viável. (13)

Assim, na pasta cmd_soap, precisamente na pasta src, encontram-se os ficheiros que implementam a aplicação. No ficheiro cmd_soap_msg.rs, encontram-se as funções relativas às operações SOAP anteriormente mencionadas. Já no ficheiro main.rs, encontra-se a aplicação em si, assim como o teste total de todas estas operações.

Infelizmente, não se implementou testes individuais para cada operação e mesmo o teste total de todas as operações não compila com sucesso.

3 Técnicas de desenvolvimento seguro de *software* utilizadas

No âmbito de se evitar hipotéticos erros de programação, é fundamental convergir para a minimização do número de vulnerabilidades de segurança não intencionais a nível de código.

Para esse efeito, pode-se recorrer à definição de padrões de programação - selecção de linguagens, estruturas e bibliotecas mais apropriadas e seguras. Para além disso, é imprescindível a garantia de uso adequado destes padrões e revisão manual do código.

De notar que se pretendia implementar mais técnicas de desenvolvimento seguro de *software*, para além das apresentadas abaixo, tais como utilizar apenas funções seguras e manipular dados com segurança, de forma a que os *inputs* fossem verificados da melhor forma possível e o código em si fosse robusto. Contudo, como não se desenvolveu destreza face à linguagem RUST, tal não foi possível. Apresenta-se, assim, de seguida, as adoptadas.

3.1 Estabelecer padrões e convenções de programação

- Informar toda a equipa de desenvolvimento acerca de tecnologia a adoptar, padrões, convenções e reutilização de programação;
- Investir na reutilização de recursos;
- Inicializar a ponderação de testes e consequente validação.

3.2 Lidar com erros

A existência de erros é inevitável, mesmo com a realização de testes de validação. Por esse motivo, é necessário que um sistema seja capaz de manipular e reagir, da melhor forma possível, a cenários imprevisíveis, apresentando uma mensagem de erro, por exemplo. Posto isto, é recomendado utilizar manipuladores de erro genéricos ou manipuladores de excepção para cobrir erros imprevistos.

De referir que a notificação do erro ao utilizador não deve relevar detalhes técnicos do problema, devendo esta ser genérica.

Nesse sentido, teve-se em consideração os hipotéticos erros que poderiam surgir em cada função. Um exemplo da prática desta técnica ocorre na função get_wsdl, uma vez que se o *input* não for 0 ou 1, é necessário reportar esse erro.

4 Ferramentas e indicadores de qualidade de *software* utilizados

Na maioria dos casos, a validação de *software* parece ser a fase mais dispendiosa e complexa de todo o processo de desenvolvimento de *software*, representando mais de metade do custo total.

As métricas de *softwares* possibilitam realizar uma das actividades mais fundamentais do processo de gestão de projectos - o planeamento. A partir deste, pode-se identificar a quantidade de esforço, de custo e das actividades que serão necessárias para a realização do projecto.

Do ponto de vista de medição, podem ser divididas em duas categorias: medidas directas e indirectas. Pode-se considerar como medidas directas do processo de engenharia de *software* o custo e o esforço aplicados ao desenvolvimento e manutenção do *software* e do produto, a quantidade de linhas de código produzidas e o total de defeitos registados durante um determinado período de tempo. Porém, a qualidade e a funcionalidade do *software*, ou a sua capacidade de manutenção, são mais difíceis de serem avaliadas e só podem ser medidas de forma indirecta.

A interpretação abstracta é uma ferramenta que permite fazer medições de forma indirecta . A interpretação abstracta formaliza a ideia de que isto é possível com um determinado nível de abstracção, onde os detalhes envolvendo a semântica e a especificação são ignorados. Um exemplo da abstracção de semântica é a lógica de Hoare, enquanto que os exemplos de abstracção de especificação são o invariante, correcção parcial e total.

Os métodos formais são interpretações abstractas com maneiras diferentes de abstrair as semânticas. Em todos os casos, as semânticas abstractas têm de ser escolhidas para serem representadas por computadores.

- Em *model-checking*, a abstracção de semântica é fornecida manualmente pelo utilizador na forma de um modelo finito de execuções do programa. Em alguns casos, estas execuções podem ser computadas automaticamente.
- Nos métodos dedutivos (*deductive methods*) a abstracção semântica é especificada por condições de verificação e têm de ser fornecidas pelo utilizador na forma de propriedades indutivas (propriedades que se mantêm verdadeiras à medida que o programa é executado) satisfazendo estas condições de verificação. Algumas propriedades indutivas podem ser calculadas automaticamente, através de análise estática.
- Em análise estática, a abstracção de semântica é calculada automaticamente graças a aproximações predefinidas, também possivelmente parametrizadas pelo utilizador.

Das ferramentas analisadas no segundo projeto não foram encontradas ferramentas que fizessem testes automáticos/abstractos ao *software* na linguagem RUST. No entanto, a *tool* Cargo permite fazer testes unitários desenvolvidos pelo utilizador, sendo, por isso, a interpretação abstracta feita por nós.

As medidas diretas utilizadas foram o número de linhas de código não comentadas (NCLOC: *Non-Comment Lines of code*), número de funções (NOF: *Number of functions/methods*).

O número de funções desenvolvidas é 9 e o número de linhas não comentadas é 213 linhas não comentadas, no geral. Relativamente às funções associadas às operações do serviço SCMD, estas são 7 e o número de linhas não comentadas é 119.

5 Modo de testar o código desenvolvido

Existem muitas formas de se testar um *software*. Mesmo assim, existem as técnicas que sempre foram muito utilizadas em sistemas desenvolvidos sobre linguagens estruturadas, que ainda hoje têm grande valia para os sistemas orientados a objetos. Apesar de os paradigmas de desenvolvimento serem completamente diferentes, o objectivo principal destas técnicas continua a ser o mesmo - encontrar falhas no *software*.

White-box testing avalia o comportamento interno de uma componente de *software*. Esta técnica trabalha diretamente sobre o código fonte do componente de *software* para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos, teste de caminhos lógicos, código nunca executado.

Os aspectos avaliados nestes testes dependerão da complexidade e da tecnologia que determinarem a construção do componente de *software*, cabendo portanto avaliação de mais aspectos que os citados anteriormente. A pessoa que realizou o teste tem acesso ao código fonte da aplicação e pode construir códigos para efectuar a ligação de bibliotecas e componentes. Este tipo de teste é desenvolvido analisando o código fonte e elaborando casos de teste que cubram todas as possibilidades do componente de *software*. Dessa maneira, todas as variações relevantes originadas por estruturas de condições são testadas.

Também se enquadram nessa técnica testes manuais ou testes efetuados com apoio de ferramentas para verificação de aderência a boas práticas de codificação reconhecidas pelo mercado de *software*. *White-box testing* é recomendado para as fases de teste de unidade e teste de integração, cuja responsabilidade principal fica a cargo dos *developers* do *software*, que por sua vez conhecem bem o código fonte produzido.

Existem outras técnicas como *Black-box testing*, em que a pessoa a realizar o teste não tem qualquer acesso ao código fonte, e *Grey-box testing*, em que o utilizador a realizar o teste tem acesso parcial ao código fonte. Não se detalhará estes últimos, pois não foram utilizados e sendo este projecto de pequena escala, todas as pessoas têm acesso ao código fonte.

Os testes manuais requerem normalmente mais trabalho e mais recursos do que os testes automáticos. Requerem também o mesmo investimento de tempo sempre que se pretender realizar o mesmo teste.

Por outro lado, os humanos têm uma maior facilidade de aprendizagem e conseguem evoluir na identificação de falhas e problemas.

Estes testes manuais são teste unitários, em que testam as menores unidades de *software* desenvolvidas (pequenas partes ou unidades do sistema). O universo alvo desse tipo de teste são as métodos, classes ou mesmo pequenos pedaços de código. Assim, o objetivo é o de encontrar falhas de funcionamento dentro de uma pequena parte do sistema funcionando independentemente do todo.

Posto isto, os testes que iriam ser realizados, seriam feitos manualmente e utilizando a técnica *White-box testing*, visto não existir uma *framework*, que consiga criar testes automatizados na linguagem RUST. Estes testes unitários seriam desenvolvidos tendo em conta a robustez da API disponibilizada, ou seja, seria testes em que validariam tipos eou campos recebidos da resposta à chamada da API.

6 Considerações finais

Ao longo do presente projeto, surgiram diversos entraves relativos à programação em linguagem RUST. Não só pela falta de agilidade face a esta linguagem, por parte de todos os membros do grupo, mas também pelo facto de parecer não existir uma *library* intuitiva e bem documentada do protocolo SOAP, nesta linguagem.

Perante isto, todo o processo de desenvolvimento foi complexo, uma vez que a linguagem RUST parece não ter uma comunidade tão consistente, quando comparada a outras linguagens como Python, C ou Java, por exemplo, pelo que os resultados das pesquisas realizadas acerca desta linguagem são escassas, o que torna a sua interiorização e manipulação difíceis.

De referir que, apesar de não se ter conseguido obter uma proposta funcional, nem completa perante o que era suposto, acredita-se que se inferiu o objetivo do trabalho prático e espera-se que, num futuro próximo, com um investimento de um maior período de tempo e com o aperfeiçoamento da destreza nesta linguagem, este possa ser realizado com sucesso.

7 Bibliografia

[1] eniaLabs, "The Ultimate List of Software Security Tools", disponível em https://xebialabs.com/the-ultimate-devops-tool-chest/security/?fbclid=IwAR2rx4jHC5KuP1o8ltAoQBvl4\discretionary{-}{}{}KEmNQ6GMx3Z5he05uKeyHTjpDm36oYIlE (acedido a 21 de Março de 2020)

[2] zone, "7 Free Security Tools That All Developers Will Want in Their Toolbox", disponível em https://dzone.com/articles/7-free-security-tools-that-all-developers-will-wan?fbclid=IwAR3E-9ruwzu9skAXe1x42bG37b53Toi8JPyyTUvJU_aAGa87IqwZaB-Yg0A (acedido a 21 de Março de 2020)

[3] icrosoft, "MIcrosoft Security development lifecicly Practices", disponível em https://www.microsoft.com/en-us/securityengineering/sdl/practices?fbclid=IwAR0H0_PIANS9DyKLC_hbjS7rMyuZAXuxd0mgUM2xDkp2aAk6a77EyRKdXSE (acedido a 21 de Março de 2020)

[4] yBOK, "SECURE SOFTWARE LIFECYCLE", disponível em https://www.cybok.org/media/downloads/Secure_Software_Lifecycle_KA_-_draft_for_review_April_2019.pdf (acedido a 21 de Março de 2020)

[5] BM, "Security in Development", disponível em https://www.redbooks.ibm.com/redpapers/pdfs/redp4641.pdf (acedido a 21 de Março de 2020)

[6] AFECode, "Fundamental Practices for Secure Software Development", disponível em https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf (acedido a 21 de Março de 2020)

[7] Jansen, P. (2018). The TIOBE Quality Indicator: A pragmatic way of measuring code quality, consultado a 29 de abril - https://www.tiobe.com/files/TIOBEQualityIndicator_v4_3.pdf.

[8] Polyspace Code Prover, consultado a 30 de abril de 2020 - https://www.mathworks.com/products/polyspace-code-prover.html.

[9] Abstract Interpretation - Mozilla Wiki, consultado a 02 de Maio de 2020 - https://wiki.mozilla.org/Abstract_Interpretation

[10] Abstract Interpretation - Departamento de Informática da Escola normal superior de Paris, consultado a 02 de maio de 2020 - https://www.di.ens.fr/~cousot/AI/#tth_sEc1

[11] Abstract Interpretation - Departamento de Informática da Escola normal superior de Paris, consultado a 02 de maio de 2020 - https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

[12] Abstract Interpretation - Wikipedia, consultado a 02 de maio de 2020 - https://en.wikipedia.org/wiki/Abstract_interpretation

[13] The Cargo Book, consultado a 03 de julho de 2020 - https://doc.rust-lang.org/cargo/