PDF PAdES (DSS & CMD) Signature Command Line Program

Henrique José Carvalho Faria nº82200

Departamento de Informática, Universidade do Minho

Resumo Neste trabalho foi pedido que se emulasse o programa *PDF PAdES* desenvolvido pela empresa *DeviseFutures* em *Perl*. Adicionalmente, este programa deve aplicar medidas de segurança que previnam diversos ataques que visem comprometer o funcionamento do sistema ou tirar partido do mesmo para obter informações. Assim, a ordem de trabalhos passou por realizar um estudo das vulnerabilidades a que o programa está sujeito seguindo-se o desenvolvimento do programa validando cada input recebido tendo em conta as vulnerabilidades previamente identificadas.
Este relatório começa por descrever o processo de emulação da aplicação, realizar uma sumula das vulnerabilidades possíveis e de que forma foram tratadas, explicar os testes realizados e de que forma estes são justificados. No final do relatório encontram-se os comandos para correr a aplicação.

Palavras-chave: Perl · Buffer-Overflow · String Vulnerabilitys · Integer Vulnarabilitys · Input Validation · white List · Black List

1 Criação da aplicação PDF PAdES em Perl

Para este trabalho seguiu-se a estrutura do programa original separando o corpo da aplicação. O corpo principal do trabalho foi colocado no ficheiro *signpdf_ cli.pl*, as operações sobre ficheiros, chave móvel e a ligação ao servidor *Soap*, ficaram no modulo *cmd_ soap_ msg.pm*, as operações referentes á comunicação com o servidor *Rest* colocaram-se num módulo chamado *dss_ rest_ msg.pm*, e as variáveis *APPLICATION_ ID* e *DSS_ REST* ficaram no modulo *cmd_ config.pm*. Adicionalmente foi criado um módulo para realizar testes e verificações de segurança na aplicação chamado *verifiers.pm*.

Para o corpo do programa em Perl começou-se por importar as variáveis *$APPLICATION_ ID* e *DSS_ REST* do modulo *signpdf_ config* através das subrotinas *get_ appid()* e *get_ rest()* respetivamente. Caso a variável *$APPLICATION_ ID* não esteja definida o programa termina.

Em seguida é verificado se o programa foi invocado com argumentos, caso contrário o programa também termina informando o utilizador que deve utilizar o comando *signpdf_ cli.py [-h]* para obter ajuda sobre como funciona o programa. Caso tenha sido invocado com argumentos é realizado um parser dos dados recorrendo ao módulo *Getopt::Long* que faz uso de flags para identificar inequivocamente cada variável recebida como parâmetro. Em seguida estes argumentos são colecionados num array cujas variáveis definidas serão verificadas fazendo uso das subrotinas pertencentes ao módulo *verifiers.pm*.
Caso o input passe nas verificações de segurança, é dado inicio ao processo de assinatura do pdf recorrendo á chave móvel digital.

2 Vulnerabilidades

No desenvolvimento de um software devemos sempre garantir que não divulgamos informação sobre como a nossa aplicação está construida. Durante este desenvolvimento deparei-me com 2 problemas sendo que o primeiro está relacionado com esta máxima e o segundo está relacionado com um problema do método open(). O desenvolvimento e análise das possiveis falhas do programa foi realizado com recurso a alguns livros[9,15,16,17,18].

O primeiro problema enuncia-se em seguida, "Como encerrar a aplicação com uma exceção passando uma mensagem de erro ao utilizador sem lhe revelar informação sobre o código da aplicação?". Para resolver este problema usamos a função *die*, o problema é que esta para além da mensagem fornece informação sobre a linha onde ocorreu a exceção. Felizmente caso se adicione $\backslash n$ ao final da mensagem de erro emitida pelo die este omite a informação referente á linha.

O segundo problema encontrado é referente á função *open*. Até ao ano 2000 a função open usava 2 parâmetros, um para a variável para a qual se lê e uma para o ficheiro a ler. O problema acontece caso o utilizador use um ficheiro cujo nome comece, por exemplo, com o sinal $>$, isto levará a que por exemplo caso seja dado como input o ficheiro *>/etc/passwd* nós acabamos por apagar o ficheiro de passwords do Linux. Para resolver este problema foi criada uma versão do open com 3 variáveis, uma para guardar a informação a ler do ficheiro, uma para o tipo de leitura a realizar no ficheiro e uma para o nome do ficheiro, esta nova versão corrige o problema apresentado anteriormente no entanto, acresce a este problema o facto de que caso o open use um pipe em vez de um ficheiro, ao falhar este devolve o pid do subprocesso na mensagem de erro, como queremos evitar divulgar qualquer informação sobre a aplicação usamos então a função *die* para emitir o erro sem comprometer a nossa implementação tomando o código a seguinte forma: *open(variável para leitura, modo de leitura, ficheiro a ler) or die ... .*

Nota: As restantes questões de segurança foram abordadas e tratadas num modulo perl á parte chamado ***verifiers.pm*** *criado para separar de forma legivel e explicita as sub-rotinas usadas para garantir a segurança do programa.*

Existem inúmeras vulnerabilidades a tratar para além das duas supramencionadas, nomeadamente:

1. Restrições sobre a memória
2. Neutralização do input durante a geração da página web
3. Improper Input Validation
4. Information Exposure
5. Out-of-Bounds Read
6. Neutralização de elementos especiais para comandos SQL (SQL Injection)
7. use after free
8. Integer Overflow or Wrapparound

9. XML Injection
10. OS Commands Injection
11. SQL Injection

Destas vulnerabilidades a 1ª 4ª, 5ª, 6ª, 7ª e 8ª podem ser ignoradas visto que o *Perl* trata da alocar as variáveis na memória libertando o programador da manutenção da mesma. Adicionalmente a vulnerabilidade número 10 foi tratada ao corrigir a falha decorrente do comando *open()*. Assim vamos debruçar-nos sobre as vulnerabilidades 3, 9 e 11.

2.1 Improper Input Validation

Nesta secção falaremos um pouco dos inputs e da validação realizada sobre os mesmos. A validação de inputs de uma aplicação é fulcral para o bom funcionamento da mesma, nunca devemos acreditar que o utilizador usará a aplicação da melhor forma em vez de a usar para fins nefastos.

Assim os inputs a verificar são: o número de telefone, o pin e o nome do ficheiro a assinar. Caso sejam fornecidos, o nome do ficheiro assinado e a data fornecida também serão verificados.

- Nomes dos ficheiros

  Para realizar a verificação tanto do nome do ficheiro de input como do ficheiro de output o processo aplicado foi o mesmo. Foram criadas 2 listas, uma white list com os caracters aceitaveis para constituirem o nome de um ficheiro (As White Lists são especialmente proveitosas visto que é mais facil indicar o que é aceitavel do que o que não é, em contrapartida limitamos um pouco os nomes possíveis para os ficheiros fornecidos) e uma black list onde removemos algumas hipóteses aceitáveis na white list mas que não podem ser dados como input do nome do ficheiro que são as flags usadas nos inputs do programa. Convem notar que tentativas de inserção de vários comandos através da adição de *;* ou de pipes com o caracter | não funcionam pois não pertencem á lista de caracters permitidos pela white list.

- Número de telefone

  No caso do número de telefone desenvolvemos 2 regex sendo que um funciona para números internacionais e nacionais e um que funciona apenas para números nacionais, os respetivos regex apresentam-se em seguida:

  - */^\+[0-9]{1,3} [0-9]{4,14}$/*
  - */^\+351 [0-9]{9}$/*

Como a chave móvel digital para a qual a aplicação se destina normalmente está associada a números de telemovel portugueses mantivemos o segundo regex embora tenhamos deixado em comentário o primeiro regex caso pretendamos estender a aplicação a números estrangeiros. A razão de escolhermos apenas números nacionais prende-se com a escolha de implementar uma segurança com granularidade mais fina visto que o número de dígitos de telemovel varia de país para país e o indicativo também.
Nota: Convém notar que, nas expressões regex, são usados por vezes 2 simbolos, o ^ no inicio do regex e o $ no fim. Estes simbolos indicam ao perl que o regex tem de corresponder desde o inicio do input até ao final deste respetivamente, isto é, caso ambos os simbolos sejam usados o perl entende que o input a testar tem de ser totalmente formado pelo regex e, caso não seja, falha a verificação.

– PIN

O PIN é um conjunto de 4 a 8 digitos, assim, para o testar bastou um regex simples que garantisse isso: */^[0-9]{4,8}$/*.

– OTP

O OTP é verificado como sendo um conjunto de 6 digitos. Mais uma vez o regex usado é bastante simples: */^[0-9]{6}$/*.

– Process ID

O ProcessID trata-se de um conjunto de 32 caracters (números e letras) que seguem um padrão especifico de formação com a seguinte caraterística: *XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX*.

Desta forma foi criado um padrão regex que permitisse verificar se o ProcessID recebido tinha 32 caracters sendo que este era constituido por conjuntos de 8, 4, 4, 4 e por fim 12 caracters separados por travessões, sendo que estes eram compostos exclusivamente por letras minúsculas e números: */^[a-z0-9]{8}-[a-z0-9]{4}-[a-z0-9]{4}-[a-z0-9]{4}-[a-z0-9]{12}$/*.

– Datetime

A Datetime trata-se de uma indicação temporal que respeita uma notação temporal reconhecida pelo servidor *Rest* com o qual a aplicação comunica: *YYYY-MM-DDTHH:MM:SS.SSSSSS*.

Para verificar se a data foi inserida respeitando a notação exigida foi criado um regex para a mesma: */^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}.\d{6}$/*. Este regex verifica se a informação é inserida, com os respetivos simbolos extra[1], na forma ano, mês, dia, hora, minutos, segundos, nanosegundos. Adicionalmente foi verificada a validade da data inserida tendo em conta os valores possíveis que o regex permite e não são válidos, como por exemplo o dia 30 do mês 2 ou a hora 27 que não existem mas que não são verificáveis pelo regex produzido. Esta verificação extra foi feita recorrendo ao módulo *Date::Manip::Range*[3] que permite fornecer uma data e verifica se esta é verdadeira.

– Response

Ao debruçarmo-nos sobre como validar a Response do servidor convêm notarmos algumas caraterísticas desta que nos podem ajudar a verificar que esta não foi alterada.

1. A resposta tem um tamanho fixo de 560 bytes independentemente do conteudo enviado para o servidor.
2. A resposta está em base 64, ou seja só possui os seguintes caracters: [a-zA-Z+/].

Á luz desta informação podemos delinear algumas verificações para mitigar o risco de ataques bem sucedidos á nossa aplicação.
A primeira verificação passa por confirmar que a Response tem o número de caracters correto, assim não é possível a adição de código extra ao conteúdo da mensagem ou truncação da mesma. A segunda verificação passa por verificar a existência apenas de caracters de base 64 na mesma. Por fim podiamos tentar verificar a existência de sintax SQL ou XML mas tal não será necessário uma vez que não é possível estas existirem visto que, em base 64, não se possuem os caracters: espaço, ponto e virgula, maior, menor entre outros necessários para as mesmas terem uma sintax correta.

– Signature

Mais uma vez á semelhança do parâmetro anterior a Signature tem um comprimento fixo e está em base 64. Assim, as verificações de segurança aplicadas apenas diferem na verificação do tamanho que passa de 560 para 512 bytes.

O principal foco da segurança na nossa aplicação foi aplicado ás strings. Os critérios usados na *white List* e na *Black List* não são muito restritivos, principalmente porque as strings são maleáveis e o utilizador pode dar o nome que quiser

[1] Simbolos extra:
- (separa a informação da data anual).
T(separa a informação da data anual da informação horária).
: (separa a informação horária).

ao documento que pretende utilizar com a aplicação. Desta forma é necessário ter atenção a utilizadores mal intencionados que pretendam usar os critérios laços de filtragem de input para fins diferentes daquele para o qual a aplicação foi feita.

2.2 OS Injection

O Perl é relativamente suscetivel á injeção de comandos do sistema operativo, isto permite utilizar pipelines com o caracter | como input ou comandos seguidos separados pelo caracter *;*. Para ambos os casos existe uma solução (White/Black List) que apesar de restringir a liberdade do cliente de nomear os seus ficheiros recorrendo aos caracters | e *;* garante que estes ataques não ocorrem, o que do ponto de vista de uma maior qualidade na segurança da aplicação é o ideal.

2.3 XML Injection

Nesta subsecção vamos tratar de qualquer tentativa de injeção de código *XML* na nossa aplicação.
Para lidar com tentativas de injeção de código *XML* através de um input foi criada uma sub-rotina chamada *xmlInjection* no modulo *verifiers.pm*. Nesta sub-rotina para evitar eventuais tentativas de injeção de código *XML* foi realizado um regex com a forma: */<[a-zA-Z]*(>[^ (<\/)]*</[a-zA-Z]*|\/)?>/* .
Este regex permite realizar match entre elementos desta linguagem através de sintaxe conhecida detetando padrões como *<qualquer coisa> ... </qualquer coisa>* ou *<qualquer coisa/>*.

2.4 SQL Injection

Para tratar eventuais tentativas de injeção de código *SQL* através das variáveis, foi definida uma sub-rotina chamada *sqlInjection* que possui um array de palávras chave usadas na syntax *SQL* que são comparadas através de um regex com os argumentos. Caso seja detetado num argumento uma palavra pertencente á syntax *SQL* o programa emite uma mensagem de erro a avisar que detetou uma tentativa de SQL injection.

2.5 Certificados Falsos

Um problema quando se lida com certificados prende-se com a validação dos mesmos. De forma a contornar este problema recorreu-se ao módulo *LWP::UserAgent*[6], este fornece uma opção por defeito de verificação automática do servidor e da sua legitimidade chamada *verify_hostname*. Assim, são escolhidos protocolos seguros e é assegurado que nos ligamos a um servidor que possui um certificado válido.

2.6 Debug

De forma a auxiliar a criação do programa foi implementado o modo debug para que se pudessem ver os envelopes enviados e recebidos bem como os respetivos headers e conteúdos. Para isso utilizou-se a biblioteca *LWP::ConsoleLogger*[7] que permite realizar o nosso objetivo de forma facil, rápida e segura invocando a sub-rotina *Everywhere()* sem argumentos.

3 Testes

Esta secção divide-se em duas subsecções, a primeira diz respeito a refactoring e identificação de Code Smells[2][8,9] e a segunda prende-se com testes feitos á aplicação para testar a sua robustez.

3.1 Refactoring e Code Smells

3.1.1 Devel::Cover

Para realizar o processo de code coverage foi usado o módulo *Devel::Cover*[10], este módulo permite correr um programa com os respetivos argumentos e monitorizar que partes do código foram usadas ou não foram usadas, criando posteriormente um relatório em html que permite visualizar de uma forma mais clara o desempenho do programa.

Para utilizar o *Devel::Cover* com o nosso programa e posteriormente criar o relatório pretendido basta correr os seguintes comandos pela ordem apresentada:

- perl -MDevel::Cover signpdf_cli.pl -u '+351 XXXXXXXXX' -p XXXX -infile teste.pdf -d
- cover

Após o programa terminar de executar podemos verificar que foi criada uma base de dados *cover_db* e que a informação da mesma foi compilada num relatório html. O resultado de correr o teste[3] é apresentado em seguida:

file	stmt	bran	cond	sub	time	total
cmd_soap_msg.pm	85.5	56.6	n/a	95.6	8.4	82.6
dss_rest_msg.pm	95.3	80.0	n/a	100.0	59.7	93.9
signpdf_cli.pl	90.5	56.2	33.3	94.1	31.8	82.1
signpdf_config.pm	100.0	n/a	n/a	100.0	0.0	100.0
verifiers.pm	91.1	57.6	n/a	100.0	0.0	75.2
Total	89.3	58.8	33.3	96.6	100.0	82.8

Figura 1: Resultado de correr o programa utilizando o módulo Devel::Cover

Analisando a imagem anterior, verificamos que existem várias colunas na tabela, cada coluna tem o seguinte significado da esquerda para a direita respetivamente: número de linhas do código utilizadas, nº de branches(if/else) utilizados[4], percentagem das condições (as condições são compostas por elementos

[2] Code Smell: característica do código fonte ou do programa que pode indicar um problema mais grave.

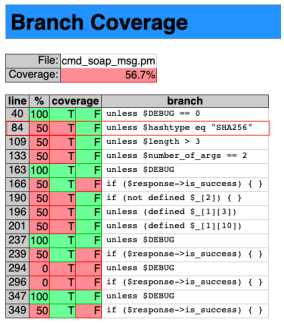
[3] Note-se que a utilização do -d não é obrigatória mas foi usado para cobrir a maior parte do código possível.

[4] O número ótimo de branches utilizados é de 50% uma vez que quando o código chega a um if then else segue apenas 1 dos caminhos possíveis, no entanto existem partes do código que possuem apenas um if ou que possuem sub-rotinas que não foram chamadas que possuem ifs, estes ifs contam para esta percentagem, fazendo o valor final subir ou descer respetivamente.

conjugados com as palavras *and* e *or*) avaliadas de forma positiva ou negativa, percentagem de sub-rotinas usadas, tempo em segundos que as sub-rotinas de cada ficheiro demoraram a executar.

Ao analisarmos os ficheiros para verificarmos que "statements"não estão a ser utilizados no código podemos verificar que estes se encontram todos dentro de condições if then else, sendo que como o programa correu bem, os statements não utilizados se encontram dentro dos else. Analisando a utilidade dos "statements"dentro de cada else constatamos que na sua maioria se tratam de comandos *die* com informação sobre o porquê do programa ter sido interrompido naquele estágio, jutificando a sua permanência.

Passando agora á análise dos branches do programa, devemos analisar para cada ficheiro se os branches são necessários, ou seja se as condições para que se siga por cada um dos caminhos das biforcações se manifestam. Tomemos por exemplo o relatório referente ao ficheiro *cmd_ soap_ msg.pm*.



Branch Coverage

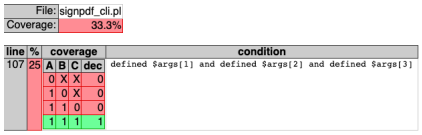
File:	cmd_soap_msg.pm
Coverage:	56.7%

line	%	coverage		branch
40	100	T	F	unless $DEBUG == 0
84	50	T	F	unless $hashtype eq "SHA256"
109	50	T	F	unless $length > 3
133	50	T	F	unless $number_of_args == 2
163	100	T	F	unless $DEBUG
166	50	T	F	if ($response->is_success) { }
190	50	T	F	if (not defined $_[2]) { }
196	50	T	F	unless (defined $_[1][3])
201	50	T	F	unless (defined $_[1][10])
237	100	T	F	unless $DEBUG
239	50	T	F	if ($response->is_success) { }
294	0	T	F	unless $DEBUG
296	0	T	F	if ($response->is_success) { }
347	100	T	F	unless $DEBUG
349	50	T	F	if ($response->is_success) { }

Figura 2: Relatório de branches do ficheiro cmd_soap_msg.pm

Após verificarmos a validade de cada branch, podemos observar que há um branch que não faz sentido porque verifica se um dado valor foi recebido como input dessa subrotina sendo que esse valor é passado de forma estática (esse branch foi assinalado a vermelho para facilitar o seu reconhecimento). Podemos então apagar esse branch uma vez que não tem utilidade no código. O mesmo processo foi aplicado a cada ficheiro, no entanto não foi mostrado visto que o tratamento é homólogo.

Podemos ainda ver, na figura1, que existe uma condição no código. O módulo Devel:Cover cria também um relatório para as condições do programa como se pode ver na imagem abaixo.



File:	signpdf_cli.pl
Coverage:	33.3%

line	%	coverage				condition
107	25	A	B	C	dec	defined $args[1] and defined $args[2] and defined $args[3]
		0	X	X	0	
		1	0	X	0	
		1	1	0	0	
		1	1	1	1	

Figura 3: Relatório de condições do ficheiro signpdf_cli.pl

Podemos verificar que são mostradas as vária verificações que podem ser levadas a cabo e os respetivos resultados. Estes resultados são todos possíveis porque até aquele ponto não existe verificação se os 3 elementos que o cliente tem de fornecer ao programa são realmente fornecidos.

Concluida a verificação de existência de código inútil, confuso, muito extenso ou duplicado foi apenas encontrado o if que fazia a verificação de um elemento estático do código.

3.1.2 Perl::Critic

Após serem removidos os code Smells do código usou-se uma biblioteca chamada *Perl::Critic*[11] que avalia o código fonte de acordo com as diretrizes do livro Perl Best Practices, além de avaliar outras métricas, como a complexidade ciclomática. Este módulo permite escolher 5 tipos de severidade de problemas no código, podendo as falhas mais severas representar problemas que permitam que um utilizador mal intencionado se aproveite do nosso programa e as falhas mais ligeiras apenas questões de legibilidade do código. Por esta razão começamos por definir uma severidade de grau 4, desta forma apenas foram apresentados os erros mais graves, de severidade 5, a vermelho e os segundos mais graves, de severidade 4[5], a laranja.

[5] Por questões de simplicidade de apresentação apenas é apresentado o resultado da aplicação do módulo ao ficheiro signpdf_cli.pl, no entanto ressalva-se que o mesmo processo foi aplicado aos restantes ficheiros do programa.

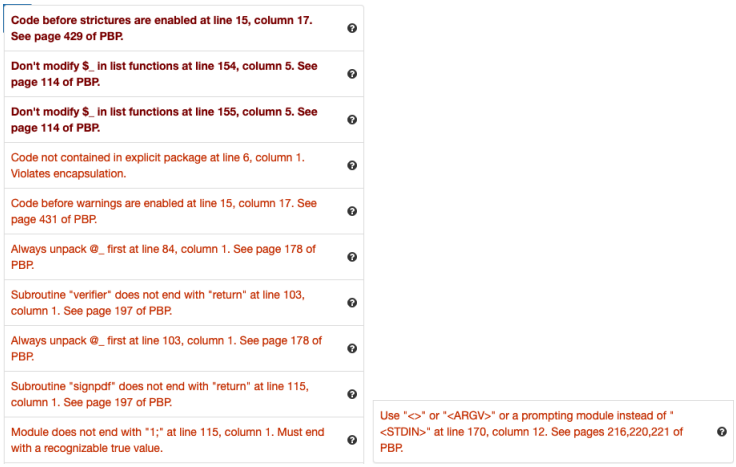


Figura 4: Relatório *Perl::Critic* de gravidades 4 e 5 do ficheiro signpdf_cli.pl

As 3 primeiras notificações apresentadas correspondem a problemas críticos no código perl que foram resolvidos adicionando, após a adição do último módulo necessario, *use strict;* para a primeira notificação e para as restantes duas bastou remmover o map que estava a ser usado para modificar uma lista de certificados aplicando um regex substituindo-o pela sub-rotina *apply()* do módulo *List::MoreUtils*[12]. As restantes notificações possuem severidade 4.

A primeira notificação refere-se á não identificação do código atual como um package, notificação essa que foi prontamente resolvida indicando na primeira linha do código *package signpdf_ cli;*. A segunda notificação desapareceu com a adição do "use strict;"utilizado para remover a primeira notifiação de severidade 5.

Duas das notificações referiam que se trata de uma boa prática copiar o array recebido por uma sub-rotina para uma variável extra e utilizar esta, uma vez que caso o array original seja alterado dentro da sub-rotina, essa alteração refletir-se-á na sub-rotina que a invocou. Adicionalmente 3 outras notificações foram resolvidas adicionando *return 1;* ao final de todas as sub-rotinas que ainda não o possuiam e *1;* no final do ficheiro, visto que os ficheiros em perl devem acabar com um valor de verdade.

Por fim o uso de *<STDIN>* foi substituido pelo uso da sub-rotina *prompt* pertencente ao módulo *IO::Prompt*[13].

Como todas as notificações para uma análise de gravidade 4 pareciam importantes, decidi diminuir o filtro de gravidade da análise do ficheiro para 3 e o resultado obtido pode-se ver na imagem abaixo.

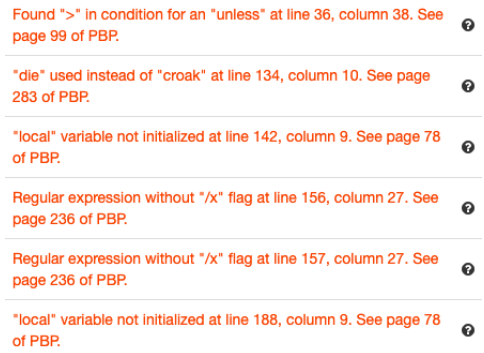


Figura 5: Segundo relatório *Perl::Critic* de gravidade 3 do ficheiro signpdf_cli.pl

Estes erros são meramente para melhorar a simplicidade aciclomática do código, ou a legibilidade do mesmo, por esse motivo apliquei as mudanças requeridas ao ficheiro.

A primeira notificação referia que ao usar o simbolo de $>$ dentro de um unless implicava uma dupla negativa, assim foi modificado o código, trocando o unless por um if.

A segunda notificação apenas pretende assinalar que quando nos deparamos com um erro de input por parte do utilizador, não devemos usar o comando *die* que serve para reportar erros da aplicação mas o comando *croak* que serve explicitamente para assinalar que o erro foi de terçeiros não da aplicação.

A terceira e quinta notificações dizem respeito á legibilidade do código, apesar de quando se declara uma variável sem lhe atribuir qualquer valor esta ser indefinida por defeito, devemos indicar explicitamente isto atribuindo-lhe o valor *undef*.

Por fim é referido que a expressão $s/\backslash s*-\!-\backslash s*BEGIN\ CERTIFICATE\backslash s*-\!-\backslash s*//$ é demasiado grande e dificil de ler e deve ser dividida para que se possam acrescentar comentários a cada parte da divisão para que se perceba o que se está a pretender obter com aquela parte do regex, no entanto, este erro deve-se á adição dos $\backslash s*$ algo que a meu ver não influencia a legibilidade, adicionalmente separar este regex tornalo-ia mais confuso do que está atualmente uma vez que este pretende representar uma string relativamtne compacta sendo os $\backslash s*$ adicionados apenas uma medida de precaução, por estes motivos estes avisos foram ignorados.

3.1.3 Test::Vars

Adicionalmente, foi criado um script perl chamado *tester.pl* que utiliza o módulo *Test::Vars*[14] para garantir que não existem variáveis por usar no programa. O resultado de correr este script sobre os ficheiros do programa é o seguinte:

```
(base) :melhoriaPerl$ perl tester.pl
ok 1 - signpdf_config.pm
# checking signpdf_config in signpdf_config.pm ...
ok 2 - cmd_soap_msg.pm
# checking cmd_soap_msg in cmd_soap_msg.pm ...
ok 3 - dss_rest_msg.pm
# checking dss_rest_msg in dss_rest_msg.pm ...
ok 4 - verifiers.pm
# checking verifiers in verifiers.pm ...
```

Figura 6: Segundo relatório *Perl::Critic* do ficheiro signpdf_cli.pl

Como se pode verificar não existe a presença de variáveis por utilizar nos ficheiros do programa.

3.1.4 SonarQube

Após todo o trabalho levado a cabo para corrigir qualquer falha no código do programa, utilizei o SonarQube[4] com uma extensão para Perl[5] para confirmar mais uma vez que o código está de facto bem escrito e não possui duplicações ou outros problemas. O resultado de correr este programa de validação e deteção de Code Smells sobre o nosso programa pode ser visualiado na figura a baixo apresentada.

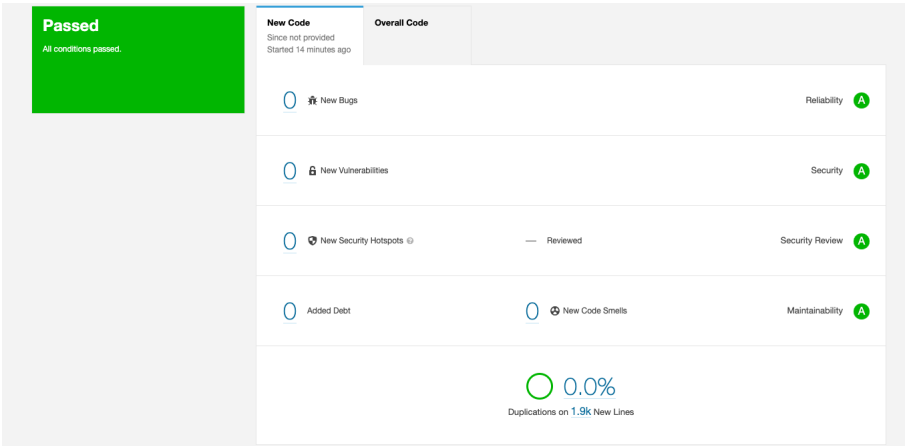


Figura 7: Relatório gerado pelo SonarQube sobre todos os ficheiros do programa

Claramente o programa não possui mais code Smells a serem retirados. Adicionalmente podemos verificar que, de facto, termos utilizado severidade de nivel 3 na análise recorrendo ao módulo *Perl::Critic* foi a escolha certa uma vez que o addOn usado no SonarQube para verificar o código perl apresenta uma utilização refinada de vários módulos[10,11] e regras de livros perl[9,15,16,17].

3.2 Testes

Para realizarmos testes sobre a aplicação, temos de verificar de que forma a podemos comprometer. A forma que temos de tentar realizar um ataque é através da inserção de inputs maliciosos, que podem ser introduzidos quando se passam argumentos no inicio programa ou no seu decorrer ou então quando o programa recebe uma resposta do servidor. Note-se que apesar de os inputs serem recebidos, todos sem exceção são enviados a alguma sub-rotina do módulo *verifiers.pm* para verificação. Assim, para testar a robustez do programa basta tentar obter um resultado inesperado nos testes do módulo *verifiers.pm* quando este deveria alertar para um erro de input, para este fim realizamos um fuzzing com recurso ao módulo *Test::LectroTest*[19]. Este módulo permite definir propriedades, propriedades essas que definem o tipo de inputs a passar ás funções a testar e o tipo de resposta esperado para esses inputs, adicionalmente, podemos criar uma mensagem informativa sobre a propriedade que está a ser testada.

Cada um dos inputs das propriedades supramencionadas pode ser definido como um tipo de variável, sendo que o módulo gerará esse input automáticamente ou alternativamente é possível criar um gerador para o input declarado, isto dá jeito quando, por exemplo, ao invés de uma string aleatória queremos uma string que contenha apenas letras minúsculas.

Foi por isso criada uma propriedade para cada verificação individual realizada no módulo *verifiers.pm* que é testada sobre os inputs fornecidos e para cada input foi criado um gerador que permite testar um parte da sub-rotina de verificação. Tomemos por exemplo os geradores, a propriedade definida e e validação dos outputs da sub-rotina *valid_ response()*, para conseguirmos definir uma propriedade sobre esta precisamos de entender o seu funcionamento primeiro.

```
sub valid_response{
    my @aux5 = @_;
    my $argument = $aux5[0];

    if(defined($argument)){
        # verifica o tamanho do input recebido
        return 0 unless length($argument) == 560;
        # verifica a validade de todos os caracteres
        return -2 unless $argument =~ /^[a-zA-Z0-9+\/]+$/;
    }
    else{
        return -1;
    }
    return 1;
}
```

Figura 8: Sub-rotina valid_response()

Esta começa por verificar se o valor recebido está definido, caso não esteja devolve -1, o que permitirá ao corpo principal do programa escrever a mensagem informativa: *"Response not defined!"*, caso a variável esteja definida temos 3 opções, a primeira é que a variável não tem o comprimento certo, isto pode ser resultado de uma interseção da mensagem por parte de terçeiros e consequente alteração do tamanho da mesma quer por truncação quer por adição, a sub-rotina prontamente devolve o código de erro 0 que permite mostrar ao utilizador a mensagem *"Illegal length found on Response!!"* . A segunda hipótese prende-se com a alteração dos valores da mensagem de forma a inserir código com fins nefastos, assim é realizada uma verificação que garante que os caracters da mensagem são todos de base 64, caso não sejam esta sub-rotina devolve o valor de erro -2 que permite mostrar a seguinte mensagem de erro ao utilizador: *"Wrong charaters on Response!!"*. Por fim temos a terçeira opção que assume que ao passar pelas várias validações o input é o original e portanto devolve o código 1 que permite que o programa avance. Assim, o nosso objetivo é fornecer um input que supostamente deva falhar uma verificação ou passar todas e cujo código devolvido não seja o esperado. O código desenvolvido com este intuito apresenta se em seguida.

```
Property {
    ##[
        x1 <- $responseGen,
        x2 <- $minResponseGen,
        x3 <- $maxResponseGen,
        x4 <- $wrongResponseGen
    # ]##
    verifiers::valid_response($x1) == 1;
    verifiers::valid_response($x2) == 0;
    verifiers::valid_response($x3) == 0;
    verifiers::valid_response($x4) == -2;
    verifiers::valid_response(undef) == -1;

}, name => "valid_response's output is 1 for any response given with length 560 and only contain
```

Figura 9: Propriedade que testa outputs da sub-rotina valid_response()

Com se pode ver na imagem acima temos dentro da propriedade 4 inputs declarados com a sintax do módulo *Test::LectroTest* e para cada input temos uma chamada á sub-rotina a testar. Os geradores destes inputs são apresentados em seguida bem como qual a propriedade que o input gerado pretende testar.

1. String(charset=>"A-Za-z0-9+/", length=>[560])
   Este gerador gera uma mensage em base 64 que deve passar todos os testes de validação de input. A validação deve retornar sempre 1.

2. String(charset=>"A-Za-z0-9+/", length=>[561,])
   Este gerador gera mensagens cujos caracters são válidos em base 64 mas cujo tamanho é superior ao esperado. A validação deve retornar sempre 0.

3. String(charset=>"A-Za-z0-9+/", length=>[1,559])

Este gerador gera mensagens cujos caracters são válidos em base 64 mas cujo tamanho é inferior ao esperado. A validação deve retornar sempre 0.

4. String(charset=>"[&%$#@"!?',;.:-_ªº~^\|]", length=>[560])
   Este gerador gera mensagens cujos caracters são inválidos em base 64 mas cujo tamanho é o esperado. A validação deve retornar sempre -2.

Adicionalmente testaram-se também as respostas da validação a um input indefinido que, como previsto, retornou -1.

O mesmo tipo de raciocinio foi aplicada a cada sub-rotina do módulo *verifiers.pm* e, no fim, as propriedades desenvolvidas foram aplicadas 10000 vezes, ou seja um número de vezes significativo para minimizar as hipóteses de terem sido gerados apenas inputs que favoreçam o intuito de cada propriedade e as avaliem como corretas. O resultado apresenta-se em baixo.

```
1..8
ok 1 - 'valid_number's output is 1 for any number given in the range 900000000 and 999999999 and 0 for
 any others.
' (10000 attempts)
ok 2 - 'valid_pin's output is 1 for any pin given with lengths between 4 and 8 and 0 for any others le
ngths.
' (10000 attempts)
ok 3 - 'valid_file's output is 1 for any pdf given and 0 for any other type of file or atempt of pipel
ine.
' (10000 attempts)
ok 4 - 'valid_datetime's output is 1 for any date given that is from the current year and from the nex
t one and 0 for any other.
' (10000 attempts)
ok 5 - 'valid_otp's output is 1 for any otp given with lengths between 4 and 8 and 0 for any other len
gths.
' (10000 attempts)
ok 6 - 'valid_processId's output is 1 for any process_id given with the propper values formation (8,4,
4,4,12) and right characters(a-z0-9) and 0 for any other formation or diferent characters.
' (10000 attempts)
ok 7 - 'valid_response's output is 1 for any response given with length 560 and only containing Base 6
4 characters and 0 for any other lengths or characters.
' (10000 attempts)
ok 8 - 'valid_Signature's output is 1 for any Signature given with length 512 and only containing Base
 64 characters and 0 for any other lengths or characters.
' (10000 attempts)
```

Figura 10: Resultado das 10000 verificações utilizando cada uma das propriedades definidas

4 Como correr o programa

4.1 Módulos

Antes de podermos correr o programa precisamos de ter os módulos requiridos pelo mesmo instalados. Para instalar os módulos necessários pode-se utilizar uma ferramenta chamada *cpanm*[20]. Pode-se descarregar esta ferramenta para linux com o comando de terminal *sudo apt install cpanminus*.
Após instalar a ferramenta deve-se garantir que se têm os seguintes módulos instalados[6]:

1. Crypt::OpenSSL::X509
2. DateTime
3. Digest::SHA
4. MIME::Base64
5. Getopt::Long
6. POSIX
7. List::MoreUtils
8. Carp
9. FindBin
10. IO::Prompt
11. REST::Client
12. XML::Compile::WSDL11
13. XML::Compile::SOAP11
14. XML::Compile::Transport::SOAPHTTP
15. Encode
16. Bit::Vector
17. HTTP::Request
18. HTTP::Parser
19. Log::Log4perl
20. LWP::ConsoleLogger
21. strict

Agora que temos os módulos necessários instalados podemos correr o programa e para o fazer temos várias opções. Para saber qual a sintaxe pela qual o programa se rege deve-se chamar o programa seguido do argumento -h:

- perl signpdf_cli.pl -h

O programa requer sempre 3 inputs obrigatórios:

- Número de telemóvel do utilizador
- Pin da chave móvel digital
- Nome do pdf a assinar

[6] Nota: Para descarregar os módulos use no terminal o comando cpanm install 'nome do módulo'

Adicionalmente o programa permite ao utilizador fornecer 2 inputs extra *Output file* e *Datetime* que são respetivamente o nome a ser atribuido ao pdf assinado e a data a usar para assinar o mesmo, podendo o utilizador escolher fornecer apenas o primeiro, apenas o segundo ou ambos.
Assim, apresentam-se em seguida as várias combinações disponíveis para a utilização da app:

1. perl signpdf_cli.pl -u '+351 XXXXXXXXX' -p XXXX -infile 'nome do ficheiro'.pdf
2. perl signpdf_cli.pl -u '+351 XXXXXXXXX' -p XXXX -infile 'nome do ficheiro'.pdf - outputfile 'nome do ficheiro'.pdf
3. perl signpdf_cli.pl -u '+351 XXXXXXXXX' -p XXXX -infile 'nome do ficheiro'.pdf -datetime YYYY-MM-DDTHH:MM:SS.SSSSSS
4. perl signpdf_cli.pl -u '+351 XXXXXXXXX' -p XXXX -infile 'nome do ficheiro'.pdf - outputfile 'nome do ficheiro'.pdf -datetime YYYY-MM-DDTHH:MM:SS.SSSSSS

Caso corra a aplicação como um utilizador regular (sem debug) o resultado esperado é o seguinte:

```
(base) :melhoriaPerl$ perl signpdf_cli.pl -u '+351 9[redacted]' -p [redacted] -infile teste.pdf -outfile teste2.pdf
Introduza o OTP recebido no seu dispositivo:309125
Ficheiro assinado guardado em teste2.pdf
```

Figura 11: Resultado de correr o programa sem o modo debug

Caso corra a aplicação em modo debug poderá ver a forma dos envelopes enviados bem como a informação que circula nos seus headers e os seus conteúdos, o resultado esperado é o seguinte:

```
(base) :melhoriaPerl$ perl signpdf_cli.pl -u '+351 9[redacted]' -p [redacted] -infile teste.pdf -outfile teste2.pdf -d
>> Debug: On

POST https://cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/CCMovelDigitalSignature.svc

.---------------------------------+-------------------------------------------------------------------------.
| Request (before sending) Header | Value                                                                   |
+---------------------------------+-------------------------------------------------------------------------+
| Accept-Encoding                 | UTF-8                                                                   |
| Content-Type                    | text/xml;charset=utf-8                                                  |
| SOAPAction                      | http://Ama.Authentication.Service/CCMovelSignature/GetCertificate       |
| User-Agent                      | libwww-perl/6.45                                                        |
'---------------------------------+-------------------------------------------------------------------------'

.-------------------------------------------------------------------------------------------------------------------------------------------------.
| Content                                                                                                                                         |
+-------------------------------------------------------------------------------------------------------------------------------------------------+
| <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Header/><soapenv:Body><GetCer-                             |
| tificate xmlns="http://Ama.Authentication.Service/"><applicationId>YjgyNjM1OWMtMDZmOC00MjVlLThlYzMtNTBhOTdhNDE4OTE2                           |
| </applicationId><userId>+351 9[redacted]</userId></GetCertificate></soapenv:Body></soapenv:Envelope>                                           |
'-------------------------------------------------------------------------------------------------------------------------------------------------'
```

Figura 12: Resultado de correr o programa com o modo debug

4.2 Replicação de testes

Caso se pretendam replicar os testes realizados sobre a aplicação é necessário instalar alguns programas bem como alguns módulos do perl. Os módulos a instalar são os seguintes:

1. Devel::Cover
2. Perl::Critic
3. Test::Vars
4. Test::LectroTest

Igualmente é necessário instalar os seguintes programas e bibliotecas, os links para obter os elementos listados também são fornecidos, cada um dos elementos possui documentação para auxiliar a sua instalação e configuração:

1. SonarQube (sonarqube.org)
2. Biblioteca Perl para o SonarQube (https://github.com/sonar-perl/sonar-perl)

Após a instalação dos recursos supramencionados basta analisar o programa criado com os mesmos.

Referências

1. http://perlcritic.com/critique/file
2. https://github.com/Hack-with-Github/Awesome-Hacking
3. https://metacpan.org/pod/Date::Manip::Range
4. https://docs.sonarqube.org/latest/analysis/overview/
5. https://github.com/sonar-perl/sonar-perl
6. https://metacpan.org/pod/LWP::UserAgent
7. https://metacpan.org/pod/LWP::ConsoleLogger
8. https://blog.codinghorror.com/code-smells/
9. Martin Fowler, Kent Beck. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional; November 30, 2018.
10. https://metacpan.org/pod/Devel::Cover
11. https://metacpan.org/pod/Perl::Critic
12. https://metacpan.org/pod/List::MoreUtils
13. https://metacpan.org/pod/IO::Prompt
14. https://metacpan.org/pod/Test::Vars
15. Peteris Krumin. *Perl One-liners*. No Starch Press, Inc.
16. *Perl - Notes for Professionals*.
17. Peter Wainwright, Aldo Calpini, Arthur Corliss Simon Cozens, Juan Julián Merelo, Guervós Chris Nandor, Aalhad Saraf. *Professional Perl Programming*. Wrox Press Ltd.
18. Tom Christiansen, Nathan Torkington. *Perl Cookbook*. O'Reilly Media, Inc.
19. https://metacpan.org/pod/Test::LectroTest
20. https://metacpan.org/pod/App::cpanminus