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 compromenter 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 a que este está sujeito. Este relatório começa por descrever o processo de emulação da aplicação e posteriormente realiza uma sumula das vulnerabilidades possíveis e de que forma foram tratadas.

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, que ficou no ficheiro signpdf_cli.pl, das operações sobre ficheiros, chave móvel e ligação ao servidor SOAP, que ficaram no modulo cmd_soap_msg.pm, das operações realizadas por um servidor REST que se colocaram num módulo chamado dss_rest_msg.pm, e das variáveis APPLICATION_ID e DSS_REST que ficaram no modulo cmd_config.pm. Adicionalmente foi criado um módulo para tratamento de segurança da aplicação chamado *verifiers.pm*.

Para o corpo do programa o Perl começa 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.
Convêm referir que, ao contrário do que foi realizado no ficheiro original *signpdf_cli.py* as verificações de segurança referentes ao código das mensagens recebidas do servidor foram realizados no módulo *cmd_soap_msg.pm*.

1.1 Módulos

Para instalar os módulos necessários pode-se utilizar uma ferramenta chamada *cpanm*. 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[1]:

1. Crypt::OpenSSL::X509
2. DateTime
3. Digest::SHA
4. MIME::Base64

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

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

2 Vulnerabilidades Gerais

No desenvolvimento de um software devemos sempre garantir que não divulgamos informação sobre como a nossa aplicação está construida. Durante o deenvolvimento do código deparamo-nos com dois possíveis problemas a tratar.

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
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 de apagar o ficheiro de passwords do Linux. Para resolver este problema usamos a 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. 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 a parte chamado ***verifiers.pm*** *criado para separar de forma legivel as subrotinas usadas para segurança das subrotinas do programa principal.*

Existem enúmeras 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, para além. Assim vamos debruçar-nos sobre a vulnerabilidade 3, 9,10 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 ou 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 ficheiro
  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 segundo 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 ao inicio do input e 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: */[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[2], na forma ano, mês, dia, hora, minutos, segundos, nanosegundos.

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

– 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 be sucedidos á nossa aplicação.
A primeira verificação pasa por confirmar que a Response tem o número de caracters correto, assim não é possível a adição de código extra ao conteúdo da mensagem. 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.

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 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 a injeção de comandos do sistema operativo, isto pois permite utilizar pipelines com o caracter / como input ou comandos seguidos separados pelo caracter *;*. Para ambos os casos existe uma solução 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 subrotina chamada xmlInjection no modulo *verifiers.pm*.
Nesta subrotina 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 subrotina chamada *sqlInjection* que possui um array de palavras 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.

3 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*, 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.

4 Como correr o programa

Para correr o programa desenvolvido 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

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 X4-8 -infile 'nome do ficheiro'.pdf
2. perl signpdf_cli.pl -u '+351 XXXXXXXXX' -p X4-8 -infile 'nome do ficheiro'.pdf - outputfile 'nome do ficheiro'.pdf
3. perl signpdf_cli.pl -u '+351 XXXXXXXXX' -p X4-8 -infile 'nome do ficheiro'.pdf -datetime YYYY-MM-DDTHH:MM:SS.SSSSSS
4. perl signpdf_cli.pl -u '+351 XXXXXXXXX' -p X4-8 -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 916393650' -p 1234 -infile teste.pdf -outfile teste2.pdf
Introduza o OTP recebido no seu dispositivo:309125
Ficheiro assinado guardado em teste2.pdf
```

Figura 1: Resultado de correr o programa sem o modo debug

Caso corra a aplicação como um utilizador regular o resultado esperado é o seguinte:

Figura 2: Resultado de correr o programa com o modo debug

5 Testes

Esta secção divide-se em duas subsecções, a primeira diz respeita a refactoring e identificação de Code Smells[3] e a segunda prende-se com testes feitos á aplicação para testar a sua robustez.

5.1 Refactoring e Code Smells

Para realizar o processo de code coverage foi usado o módulo Devel::Cover, 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[4] 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 3: 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: número de linhas do código utilizadas,nº de branches(if/else) utilizados[5], percentagem das condições (as condições são compostas por elementos conjugados com as palavras *and* e *or*) avaliadas de forma positiva ou negativa, percentagem de subrotinas usadas, tempo em segundos que as subrotinas de cada ficheiro demoraram a executar.

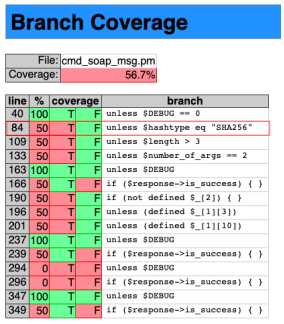
[3]

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

[5] 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 subrotinas que não foram chamadas que possuem ifs, estes ifs contam para esta percentagem como não tendo sido avaliados, fazendo o valor final descer.

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 do else constatamos que na sua maioria se tratam de comandos *die* com informação sobre o porquê do programa ter sido interrompido naquele estágio.

Passando agora á análise dos branches do programa, devemos analisar para cada ficheiro se os branches são necessários, 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

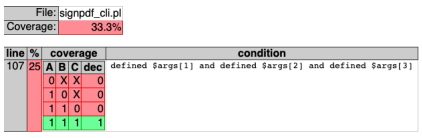
File:	cmd_soap_msg.pm
Coverage:	56.7%

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 4: 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 valor foi recebido, sendo que esse valor é passado de forma estática a essa função. 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.

Podemos ainda ver 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
		A	B	C	dec	
107	25					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 5: 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.

De facto, após a remoção de um if que se encontrava a mais no código este não apresenta mais nenhum tipo de código inutil, confuso, muito extenso ou duplicado.

Agora que removi os code Smells do código vou usar uma biblioteca chamada Perl::Critic que avalia o código fonte de acordo com as diretrizes do livro Perl Best Practices, além de outras métricas, como 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[6], a laranja.

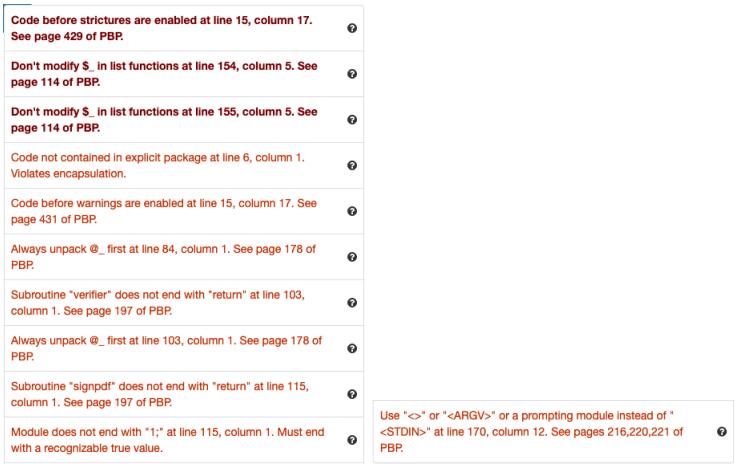


Figura 6: Relatório Perl::Critic de gravidades 4 e 5 do ficheiro signpdf_cli.pl

As 3 primeiras notificações apresentadas correspondem a problemas criticos 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 substituir o map que estava a modificar uma lista de certificados através do regex pela subrotina *apply()* do módulo *List::MoreUtils*.

[6] Por questões de simplicidade apenas é apresentado o resultado da aplicação do módulo ao ficheiro signpdf_cli.pl, no entanto o mesmo processo foi aplicado aos restantes ficheiros

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;".

Duas das notificações referiam que se trata de uma boa prática copiar o array recebido por uma subrotina, uma vez que caso este seja alterado dentro da subrotina, essa alteração notar-se-á na subrotina que a invocou. Adicionalmente 3 outras notificações foram resolvidas adicionando *return 1;* ao final de todas as subrotinas que ainda não o possuiam e *1;* o final do ficheiro, visto que os ficheiros e perl devem acabar com um valor de verdade.

Por fim o uso de $<STDIN>$ foi substituido pelo uso do *prompt* pertencente ao módulo *IO::Prompt*.

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.

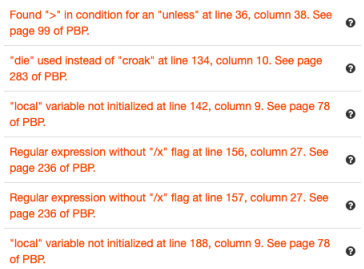


Figura 7: Segundo relatório Perl::Critic de gravidade 3 do ficheiro signpdf_cli.pl

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 um terçeiro não da aplicação.

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

Por fim é referido que a expressão $s/\s*-----\s*BEGIN\ CERTIFICATE\s*-----\s*//$ é demasiado grande e deve ser dividida para que se possão acrescentar comentários a cada parte para que se perceba o que se está a passar, 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 be mcompacta e curta sendo os $\backslash s*$ adicionados apenas uma medida de precaução.

Adicionalmente, foi criado um script perl chamado *tester.pl* que utiliza o módulo *Test::Vars* 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 8: Segundo relatório Perl::Critic do ficheiro signpdf_cli.pl

5.2 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.

Referências

1. http://perlcritic.com/critique/file