

Escola de Engenharia
Universidade do Minho

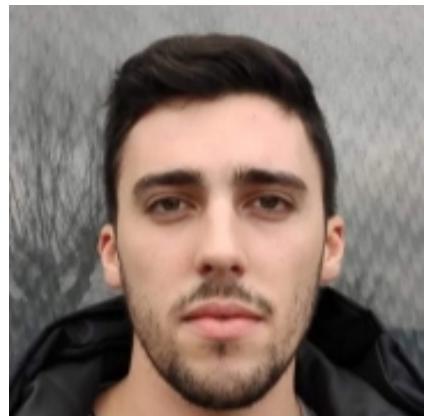
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado em Engenharia Informática
Engenharia de Segurança

- * Linguagem C
- * Projeto 1 - Tactical Threat Modeling
- * Projeto 2 - Ferramentas e técnicas de compiler warnings
- CMD-SOAP - Teste das operações do serviço SCMD**

Grupo 1



Ricardo Pereira - A73577



Tiago Ramires - PG41101

Braga, 6 de Julho de 2020

1. Introdução

VERIFICAR TODOS OS PRINTS PARA NÃO POR LÁ O NR DE TELE VERIFICAR TODOS OS PRINTS PARA NÃO POR LÁ O NR DE TELE VERIFICAR TODOS OS PRINTS PARA NÃO POR LÁ O NR DE TELE VERIFICAR TODOS OS PRINTS PARA NÃO POR LÁ O NR DE TELEVERIFICAR TODOS OS PRINTS PARA NÃO POR LÁ O NR DE TELEVERIFICAR TODOS OS PRINTS PARA NÃO POR LÁ O NR DE TELEVERIFICAR TODOS OS PRINTS PARA NÃO POR LÁ O NR DE TELE

TARDE DE MAIS!!

A área da engenharia de segurança consegue ser extremamente abrangente, abarcando inúmeras áreas de outros campos da ciência. Nos dias que correm é quase impossível não estar perto de um dispositivo que necessite de um determinado tipo de protecção, seja um sensor biométrico, um *smartphone*, um *router*, entre outros. Se se atentarem outros casos, apercebe-se que a engenharia de segurança é também aplicada em dispositivos médicos, dispositivos bancários, sendo por isso uma área extremamente requisitada para conceber e assegurar a protecção dos mais diversos sistemas que podemos encontrar no nosso dia a dia.

Contudo, apesar de existir um ramo da engenharia orientado à segurança de sistemas, é muito frequente ver-se a ocorrência de falhas extremamente básicas, muitas das vezes detectadas após ataques bem sucedidos, que implicam custos extremamente avultados para as organizações que deles foram alvo. Assim, a principal questão impõe-se: prevenir ou remediar?

Viu-se, em projectos anteriores, que é urgente a inclusão de boas práticas na construção de *software* para que este seja fidedigno e confiável, tal como muitas vezes acreditamos que é. Neste projecto, o objectivo é mesmo esse - construir um programa que reproduza as operações de assinatura com Chave Móvel Digital tendo em conta todos os aspectos de segurança mais importantes que se conhecem, sendo um deles, por exemplo, a validação dos inputs introduzidos pelo utilizador. Por se saber que a interacção do utilizador com o programa está directamente ligada à existência de vulnerabilidades, decidiu-se apostar bastante nesse campo, garantindo-se que todo o *input* obedece a determinados padrões sendo ainda bastante flexível às necessidades do utilizador.

Outro desafio que se impõe é a reprodução de código escrito em *Python* em linguagem *C*, tendo em conta que a primeira é muito mais "popular" e por esse motivo tem a vantagem de garantir mais suporte àqueles que a utilizam para programar. Por outro lado, o *C* é uma linguagem muito mais rígida, o que obriga a uma maior pesquisa para a sua utilização, implicando também mais linhas de código e por conseguinte, maior complexidade.

2. Desenvolvimento

2.1 Estudo da Aplicação

Para iniciar a construção da aplicação é necessário estudar aquilo que é suposto fazer, ou como neste caso, aquilo que a mesma já faz. Após uma fase de testes da aplicação construída em *Python* verificou-se que mesma se comportava da seguinte maneira - o utilizador pode decidir executar os vários comandos de forma independente, sendo que cada um necessita de determinados parâmetros:

- a função ***getcertificate*** obtém o certificado que está associado ao utilizador da aplicação pela entidade certificadora que se enquadra numa determinada hierarquia que mais tarde será utilizado - a mesma precisa que lhe seja dado o **número de telefone do utilizador**;

```
request_data = {
    'applicationId': args.applicationId.encode('UTF-8'),
    'userId': args.user
}
return client.service.GetCertificate(**request_data)
```

Figura 2.1: Parâmetros que a função *getcertificate* envia ao servidor.

- a função ***ccmovelsign*** devolve o identificador que é atribuído ao processo de assinatura, ao qual está associado um *OTP* que é enviado para o **número de telemóvel** que foi introduzido como parâmetro, tal como o **PIN** associado à assinatura da Chave Móvel Digital;

```
request_data = {
    'request': {
        'ApplicationId': args.applicationId.encode('UTF-8'),
        'DocName': args.docName,
        'Hash': args.hash,
        'Pin': args.pin,
        'UserId': args.user
    }
}
print(client.service.CCMovelSign(**request_data))
return client.service.CCMovelSign(**request_data)
```

Figura 2.2: Parâmetros que a função *ccmovelsign* envia ao servidor.

- a função ***ccmovelemultiplesign*** faz o mesmo que a anterior mas permite a assinatura de mais que um ficheiro;

```

request_data = {
    'request': {
        'ApplicationId': args.applicationId.encode('UTF-8'),
        'Pin': args.pin,
        'UserId': args.user
    },
    'documents': {
        'HashStructure': [
            {'Hash': hashlib.sha256(b'Nobody inspects the spammish repetition').digest(),
             'Name': 'docname teste1', 'id': '1234'},
            {'Hash': hashlib.sha256(b'Always inspect the spammish repetition').digest(),
             'Name': 'docname teste2', 'id': '1235'}
        ]
    }
}

return client.service.CCMovelMultipleSign(**request_data)

```

Figura 2.3: Parâmetros que a função *ccmovelmultiplesign* envia ao servidor.

- a função *validate_otp*, tal como o nome sugere, envia para o servidor o código que o mesmo enviou para o telemóvel do utilizador, necessitando por isso do **identificador do processo** e, obviamente, do **OTP** como parâmetros.

```

request_data = {
    'applicationId': args.applicationId.encode('UTF-8'),
    'processId': args.ProcessId,
    'code': args.OTP,
}
return client.service.ValidateOtp(**request_data)

```

Figura 2.4: Parâmetros que a função *validate_otp* envia ao servidor.

A execução independente dos comando anteriores, por si só não faz sentido, uma vez que umas necessitam de informação obtida por outras. Assim, todos os comandos podem ser executados de uma forma sequencial - é nisto que consiste a função *testall*: primeiro, obtém os certificados com os quais irá verificar a assinatura, depois inicia o processo de autenticação e assinatura, sendo devolvidos o identificador de processo e um código para o número introduzido. Este código é introduzido na aplicação e é devolvida uma assinatura que é posteriormente verificada.

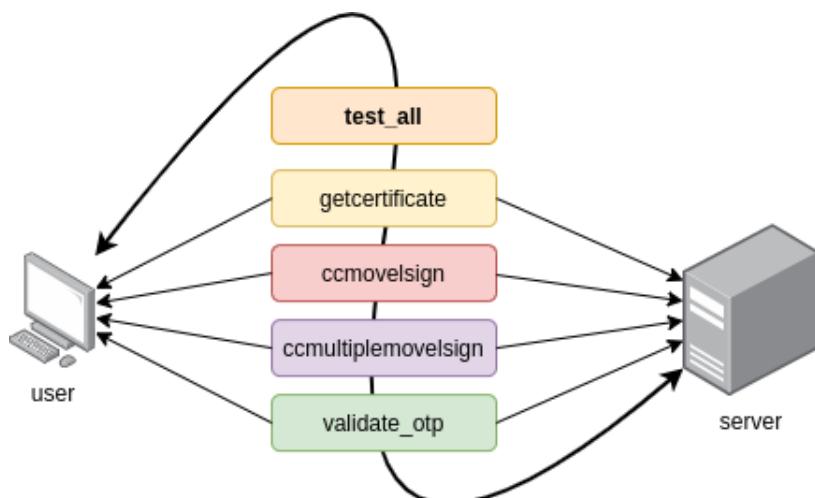


Figura 2.5: Funções do servidor utilizadas.

Devido ao facilitismo do *Python*, a análise dos argumentos é toda feita pelo módulo *argparse*, não havendo grande margem do utilizador para explorar vulnerabilidades, uma vez que a verificação do *input* cinge-se bastante à interpretação do servidor. O utilizador introduz os argumentos e o tamanho dos mesmos nunca é verificado, por exemplo, ficando o servidor encarregue de lidar com isso. Uma das validações de *input* que é feita, é por exemplo, a verificação do ficheiro, pois para se assinar o ficheiro é necessário abri-lo.

Outra parte importante do programa que motivou bastante pesquisa foi a comunicação com o servidor, feita através de mensagens *SOAP*. O programa encapsula numa mensagem deste tipo, ver figura 2.6 e os dados a enviar são definidos pelo servidor através da disponibilização de métodos que o programa tem acesso para este serviço. Em resposta ao pedido, o servidor devolve uma mensagem do tipo *SOAP*, em que o programa irá analisar e extrair toda a informação necessária.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <GetCertificate xmlns="http://Ama.Authentication.Service/">
      <applicationId>[base64Binary?]</applicationId>
      <userId>[string?]</userId>
    </GetCertificate>
  </Body>
</Envelope>
```

Figura 2.6: Mensagem *SOAP*.

No comando *testall*, a parte final consiste na verificação da assinatura. Este processo consiste na aplicação da função *hash sha256* ao conteúdo do ficheiro, na obtenção da chave pública do certificado do utilizador e na verificação da assinatura com o *digest* da mensagem a verificar, sendo que para esta verificação, é utilizado o esquema de assinaturas *RSA pkcs1 v1.5*.

```
digest = SHA256.new()
digest.update(file_content)
public_key = RSA.import_key(certs[0].as_bytes())
verifier = PKCS1_v1_5.new(public_key)
verified = verifier.verify(digest, res['Signature'])
```

Figura 2.7: Verificação da assinatura.

2.2 Construção da Aplicação

Tendo em conta tudo o que foi dito anteriormente, começou-se a construir a aplicação pela parte mais importante, a validação do *input* do utilizador, que será explicado com mais detalhe na próxima secção. No geral, a aplicação em *C* segue a arquitectura da aplicação em *Python*, tendo um módulo principal *main.c*:

- *main* - função que chama a *args_parse*;
- *args_parse* - função que comprehende os argumentos introduzidos pelo utilizador; conforme o comando que o utilizador especificou, o programa faz determinadas verificações;
- *testall* - função que agrupa todos os comandos, executando-os e retornando a validação da assinatura resultante de todo o processo.

Para além deste, existem mais dois módulos que este programa necessita para o seu funcionamento, sendo eles o *cmd_config.c* onde se define por *default*, o identificador da aplicação atribuído pela *AMA* e o módulo *cmd_soap_message.c*:

- *getcertificate*, *ccmovelsign*, *validate_otp*, *testall* - têm o mesmo comportamento das funções homónimas do programa em *Python*;
- *strreplace* - função que numa *string*, substitui uma *substring* por outra;
- *base64Encoder* - função que descodifica uma *string* codificada em *Base64*;
- *base64Decoder* - função que codifica numa *string* no formato *Base64*;
- *strcut* - função que corta uma *string* após encontrar uma *substring*, ficando com a parte inicial da *string* até à ocorrência da *substring*, inclusive;
- *WriteMemoryCallback* - função relacionada com a comunicação *CURL*, para a escrita em variável do output da comunicação;
- *to_sha256* - função que devolve o *digest* resultante após a aplicação do *sha256* a uma *string*;
- *my strdup* - função que reproduz o comportamento da função *strdup* que não é aceite pelo *C99 compliance*.

A interacção dos módulos é extremamente necessária e por esse motivo foram criados *headers* para agilizar o processo de programação, sendo colocadas nos mesmos as assinaturas das funções que são exportadas e posteriormente utilizadas noutras módulos.

Em *Python*, não é comum definirem-se estruturas de dados uma vez não há grande rigidez nos tipos de dados e também porque é possível, por exemplo, criar uma *hashtable* simplesmente criando um *array* em que os índices são *strings*. Em *C* não há essa liberdade e para se poderem guardar os argumentos e associar-lhes um significado utilizou-se uma *hashtable* da biblioteca *gmodule*. Isto facilitou bastante o acesso aos dados que são guardados durante o *parsing*.

```
GHashTable *table[NR_OPERATIONS];
```

Figura 2.8: Criação de um *array* de *hashtables* onde cada posição corresponde a uma *hashtable* com informações dos argumentos introduzidos para aquele comando.

```
g_hash_table_insert(table[gc], "user", argv[i]);
```

Figura 2.9: Inserção do número de telemóvel do utilizador na *hashtable* associada ao comando *gc*.

```
(char *) g_hash_table_lookup(table[test], "file")
```

Figura 2.10: Obtenção do nome do ficheiro introduzido com o comando *test*.

Para fazer a comunicação com o servidor, o programa *Python* utilizou um cliente *SOAP*, algo que é um pouco mais complicado em *C*. Após alguma pesquisa e tentativas para utilizar a biblioteca *gSOAP*, chegou-se à conclusão que a implementação requeria a instalação de

módulos adicionais que não foram encontrados por estarem bastante descontinuados e por serem pouco falados nas comunidades de programadores. Por esse motivo, resolveu-se adoptar uma comunicação que utiliza *CURL*, remediando-se assim o problema. Assim, as chamadas aos vários métodos da *SOAP*, são feitas através de invocações *RESTful* do tipo *POST*, em que as mensagens *SOAP* são construídas com sucessivas concatenações e determinados elementos dessa mensagem sofrem uma pequena alteração com a adição do prefixo *soapenv*, resultando numa mensagem no formato *XML* do tipo *SOAP* que irá ser enviada ao servidor no *body* do pedido.

```
strcat(xml_to_send, "<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/");
strcat(xml_to_send, base64Encoder(applicationId, strlen(applicationId)));
strcat(xml_to_send, "</applicationId> <userId>");
strcat(xml_to_send, userId);
strcat(xml_to_send, "</userId> </GetCertificate> </soapenv:Body> </soapenv:Envelope>");
```

Figura 2.11: Construção da mensagem *SOAP*.

A execução desta aplicação está totalmente dependente das respostas do servidor em questão e como tal, as funções que comunicam com o mesmo retornam uma *string* constituída por um código e pelo parâmetro que é suposto vir na resposta. O código dita a forma como o servidor lidou com o problema e para as funções em questão, uma resposta com o código 200 significa que não houve nenhum problema. Qualquer outro código implica a existência de problemas e consequentemente, a interrupção do programa.

```
if(!strcmp(strtok(otp_answer, "|"), "200")){
    signature = strtok(NULL, "|");
    printf("%s\n", signature);
}
else {
    printf("%s\n", "Erro");
    exit(EXIT_FAILURE);
}
```

Figura 2.12: Verificação do valor do código.

Resta a verificação da assinatura, que é feita com recurso a bibliotecas da área da criptografia. À semelhança do que é feito no programa em *Python*, submete-se o conteúdo do ficheiro a uma função de *hash*, extrai-se a chave pública, e verificam-se as assinaturas.

```
if(!strcmp(strtok(otp_answer, "|"), "200")){
    signature = strtok(NULL, "|");
    printf("%s\n", signature);
}
else {
    printf("%s\n", "Erro");
    exit(EXIT_FAILURE);
}
```

Figura 2.13: Verificação da assinatura.

2.3 Aspectos de Segurança

Como é sabido, a linguagem na qual o projeto deve ser construído é extremamente vulnerável a falhas de *buffer overflow*, que são a base para ataques extremamente problemáticos, como

já se pôde constatar em situações passadas. Neste projeto, o objectivo é construir uma aplicação que assine documentos mas também é suposto mostrar que formas existem de validar inputs do utilizador.

2.3.1 Comparação simples

Nos argumentos do programa que o utilizador introduz tem que existir um comando que pertença àquela lista de cinco comandos. Caso o argumento que venha a seguir ao nome do programa não seja um dos comandos disponíveis, a aplicação é abortada e esta verificação é feita com uma simples comparação de *strings*.

```
!strcmp(argv[1], "GetCertificate") || !strcmp(argv[1], "gc")
```

Figura 2.14: Verificação do comando a executar.

2.3.2 Expressões regulares

Para a validação de *input*, a ferramenta pela qual se optou, cuja utilidade é transversal a qualquer área do desenvolvimento de *software*, foram as **expressões regulares**. As mesmas permitem estabelecer determinados padrões específicos aos quais as *strings* devem obedecer não havendo assim margem para erro. Imaginemos que o utilizador deve introduzir uma *string* com quatro dígitos e introduz cinco ou introduz mesmo uma *string* de formato - a expressão regular que aceita apenas quatro dígitos, rejeita imediatamente a *string* introduzida, visto que esta não obedece à expressão criada. Este exemplo enquadra-se perfeitamente na validação do *PIN* que pode ter entre quatro e oito dígitos.

```
regcomp(&pin_regex, "[0-9]{4,8}", 0);
```

Figura 2.15: Expressão regular que define o *PIN*

Praticamente todas as validações de *input* do utilizador são feitas com base nesta ferramenta.

2.3.3 Opções do compilador

Em projetos anteriores, foi evidente a necessidade de incluir na compilação dos programas várias opções que activam a monitorização do código para que cumpra determinados *standards* vistos como boas práticas que devem ser tidas em conta durante a produção do mesmo. Agora, essas opções provam mais uma vez a sua utilidade tendo surgido várias vezes durante a construção do programa alertas de falhas no código que davam azo à ocorrência de **segmentation faults**, algo que poderia mais tarde ser explorado pelos utilizadores do programa. Por exemplo, após ter sido construído um ciclo *for* que percorria o *array* de *hashTables* surgiu o seguinte *warning*:

```
src/main.c: In function 'main':  
src/main.c:39:12: warning: iteration 5 invokes undefined behavior [-Waggressive-  
loop-optimizations]  
39 |     table[i] = g_hash_table_new(g_str_hash, g_str_equal);  
|     ^~~~~~  
src/main.c:38:2: note: within this loop  
38 |     for(i = 0; i <= NR_OPERATIONS; i++)  
|     ^~~
```

Figura 2.16: Alerta para comportamento irregular do programa.

O mesmo avisa o programador que a quinta iteração do ciclo não devia acontecer talvez porque o compilador compreendeu que não fazia sentido estar a aceder a posições do *array* não definidas. O erro é uma distração evidente e corrige-se com a alteração da comparação que está a ser feita na condição do ciclo, passando a ser uma comparação de "menor que" em vez de "menor ou igual que". Outro tipo de *warnings* que têm toda a razão de ser são os de variáveis não utilizadas, que podem causar entropia no código, ficando ainda mais complexo.

Todavia, a compilação do programa apresenta alguns *warnings* que não foram esquecidos, simplesmente contrariam propositadamente alguns *standards* de compilação. O incremento da variável *i* antes da operação em que está incluída é intencional e por esse motivo, os *warnings* em questão foram ignorados.

```
src/main.c:549:62: warning: operation on 'i' may be undefined [-Wsequence-point]
  549 |         g_hash_table_replace(table[test], argv[i], (gchar *) argv[+~i]);
          |           ^~~
```

Figura 2.17: Warning referente ao incremento da variável *i*.

Foram então ativadas as seguintes opções na compilação do programa: *ansi*, *Wunreachable-code*, *O2*, *W uninitialized*, *W unused-parameter*, *Wall* e *Wextra*.

2.3.4 Funções de leitura do *STDIN*

Existem determinadas funções que não devem ser utilizadas apesar de serem úteis ou apenas mais simples. A função *gets*, por exemplo, requer menos argumentos mas é mais permeável à ocorrência de *segmentation faults*, uma vez que não verifica o tamanho da *string* que é introduzida no *STDIN*. Com uma função praticamente igual, consegue-se ter o mesmo efeito mas com muita mais segurança: a função *fgets* que limita o tamanho da *string input*.

```
printf("Introduza o OTP recebido no seu dispositivo: \n");
if(fgets(buf, 7, stdin))
    buf[6] = '\0';
else printf("Erro na leitura do OTP\n");
```

Figura 2.18: Função *fgets* utilizada para a leitura do *OTP*.

Esta função foi utilizada para a leitura do *OTP* introduzido pelo utilizador durante a execução do programa. Caso se tivesse usado a *gets*, surgiria com certeza um *warning* durante a compilação para evitar o uso da mesma.

2.3.5 Utilização do *GDB*

Uma forma de compreender eficazmente a origem dos *segmentation faults*, passa pela utilização do *GDB*, uma ferramenta *GNU* que possibilita a interação do programador com o programa durante a execução do mesmo. Para o utilizar apenas é necessário executar o comando *make debug*. Com esta ferramenta, é possível "olhar para dentro" dos estados da máquina tal como eles existem e verificar se de facto têm o que era esperado. Como não poderia deixar de ser, a mesma foi muito útil uma vez que ajudou na compreensão dos erros que estavam por trás dos *segmentation faults*.

2.3.6 Prevenção de *Memory Leaks*

O termo *memory leaks* é bastante conhecido, principalmente por programadores que utilizam a linguagem em questão, uma vez que representa um problema bastante recorrente. Um programa que tem *memory leaks* é um programa para o qual, a determinada altura, foi aloçado espaço na memória e esse espaço nunca mais foi libertado. Ora, rapidamente se chega à conclusão que um programa que aloca muito espaço na memória deve libertá-lo assim que não precisar mais dele, caso contrário a máquina fica em esforço sem necessidade. Mais grave ainda, estes *memory leaks* podem ainda levar a um comportamento inesperado do programa, mais concretamente do conteúdo das suas variáveis.

Por todos estes motivos, sempre que tenha havido alocação de memória houve também a preocupação de libertar o espaço alocado com a utilização da função *free*, algo que aconteceu com as *hash tables* (função *g_hash_table_destroy* que implementa a função *free* para destruir as *hash tables*), por exemplo.

```
for(i = 0; i < NR_OPERATIONS; i++)
    g_hash_table_destroy (table[i]);
```

Figura 2.19: Função *g_hash_table_destroy* que liberta o espaço alocado para as *hash tables*.

2.4 Funcionamento do Programa

O programa pode ser executado de forma muito simples, uma vez que utiliza uma *Makefile*, automatizando o processo de compilação e execução. Posto isto, existem duas formas de compilar e executar o programa (ambas necessitam da *bash* posicionada na pasta onde se encontra o ficheiro *Makefile*): a primeira é escrever e executar no terminal o seguinte comando: **make run_<comando pretendido>**, mas para que corra com a informação do utilizador é necessário alterar o conteúdo da *Makefile* e colocar no comando em questão os argumentos pretendidos; a segunda é escrever e executar **make**, sendo imprimida alguma informação de compilação e após isso escrever e executar **./program <comando> <argumento 1> <argumentos 2> ...**.

- * Não é indicado arquivos e pacotes necessários nem o modo de os instalar: glib2.0, libglib2.0, phg-config, libcurl
- * Executar o programa sem parâmetros dá Segmentation Fault
- * diz que program copys -h, dí o help para a operação, mas tal não acontece
- * não valida a assinatura na operação test
- * O T9 não numérico dá Segmentation Fault - operação test

3. Conclusão

Tudo aquilo que se relatou neste documento não é mais do que um conjunto de boas práticas que desde sempre devem ser implementadas em qualquer tipo de projecto, quanto mais em *software* construído em linguagens como *C*. Verificação de tamanhos de *strings*, inclusão de opções de *warnings* na compilação, utilização de ferramentas de análise são aspectos que podem fazer toda a diferença entre *software* bom e mau.

A utilização e implementação de boas práticas na programação deve indiscutivelmente, ser uma preocupação daqueles que querem construir *software* fidedigno e funcional, visto que a inclusão de mais um linha de código que verifica uma determinada variável, pode fazer a diferença entre uma empresa ter ou não que sofrer consequências gravíssimas por falhas de *software*.

É agora mais do que evidente que os argumentos que comumente se ouvem são infundados e já se podem refutar com situações práticas. Há relatos de várias organizações que começaram a investir nesta área após verificarem a presença de erros banais nos seus sistemas, alguns que resultaram em repercussões económicas impensáveis.

A compilação de programas deve, por todos os motivos vistos anteriormente, ter em atenção todos os *errors* e *warnings* que são fundamentais para que os programas construídos sejam fiáveis e não originem erros durante a execução dos mesmos. É uma mais valia e é importante lembrar que não há qualquer custo monetário necessário para obter esses avisos, pelo que essa não pode ser uma desculpa válida. Além dessas, existem outras formas, também elas gratuitas, de verificar programas disponíveis para várias linguagens.

Para finalizar, é importante realçar a pouca informação que existe na rede acerca deste tipo de implementações tendo sido bastante difícil encontrar determinadas funções que faziam o que era suposto, como as funções de verificação da assinatura. As bibliotecas eram também elas pouco esclarecedoras o que dificultou ainda mais a conceção deste projeto. Contudo, o objetivo principal foi atingido, tendo os aspectos de segurança sido devidamente estudados e aplicados no programa em questão.