



**Ciências
ULisboa**

Faculdade
de Ciências
da Universidade
de Lisboa

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Mestrado em Engenharia Informática e Segurança Informática

RELATÓRIO

Segurança de Software / Sistemas de Software Seguros

Final Project

Grupo SS12

Rodrigo Craveiro Rodrigues (fc64370)

Diogo Serrano Sargaço (fc58252)

Kevin Alexandre Lima dos Santos (fc64874)

Professor: **Doutor Nuno Ferreira Neves**

1º Semestre Letivo 2024/2025

Dezembro 2024

Índice

<i>Lista de Acrónimos</i>	5
<i>Lista de Figuras</i>	6
2. VulnApp	7
01. Characterize the attack surface of each application. Justify your answer by relating it to the code.....	7
02. The map path application has several vulnerabilities, including several buffer overflows. How many buffer overflow vulnerabilities do you find, and where (line number) do they occur?	8
03. Describe an input to the program that exploits one of your identified vulnerabilities. ..	9
05. How should the program be modified to prevent this vulnerability? Explain how your patch is effective at preventing the exploitation of the flaw.	11
06. The mysig application may have an integer overflow flaw that allows for a buffer overflow. Could you confirm if this problem exists? Explain how the adversary could exploit the bug.	12
07. Explain how you would modify the application to prevent the flaw's exploitation.....	13
3. SSS-DB	13
09. The Show Log option of the Core Controls menu entry allows a stored XSS attack. Explain why and describe how the attack could be performed (the attack can show a popup window saying "hello").....	15
10. The option Text File Viewer of the Operations menu allows the display of a specific document about hacking (file text-file-viewer.php). This option is vulnerable to a reflected XSS attack even though the user does not write any text, and they can only choose a document from a fixed list. Explore this vulnerability to show, for example, a popup window saying "hello."	15
11. Explain why this vulnerability exists by looking into text-file-viewer.php. How could you protect yourself from this attack?	16
12. The DNS Lookup option of the Operations menu entry is vulnerable to a command injection vulnerability (file dns-lookup.php). Explore this vulnerability to get the credentials of access to the application's database (host, database name, username and	

password).	17
13. Explain why this vulnerability exists by looking into dns-lookup.php. How could you protect yourself from this attack? (HINT: look at the part of the code executed with a higher security level).	19
4. <i>Flawfinder</i>	21
14. At the folder of the tool (/home/ss/apps/flawfinder), execute the next command: ./flawfinder -m 2 ../vulnApp/mysig/* . For each error reported by the tool, indicate if it is actually a vulnerability.	21
15. Execute the same command but add the -F option (does not output False Positives) ./flawfinder -m 2 -F ../vulnApp/mysig/*	25
Compare the results with the analysis you did in the previous question. Is the tool precise in its results? Justify your analysis and answer in the report.	25
6. <i>AFL</i>	26
16. Use the AFL fuzzer tool to discover vulnerabilities in an extensive library made vulnerable by injecting a few bugs. While compiling the code, several applications are generated that call the library (see below). We will test the tiffcp application for this exercise.	26
7. <i>Large Language Models</i>	31
17. Analyze manually the two codes and determine if they are vulnerable or secure to the above mentioned flaws. Your answer should justify your conclusions carefully. Ao analisar manualmente os códigos dos ficheiros python COLE.py e LEAKS.py, podemos identificar as seguintes vulnerabilidades mencionadas:	31
18. Imagine that you are the developer of the two codes and would like ChatGPT to help you determine if they are vulnerable. It would be best if you designed the most appropriate question for ChatGPT so that it provides the most helpful response (this is called prompt engineering). In your report, you must explain the steps you have taken to create the most effective question for ChatGPT.	32
19. Ask ChatGPT to provide you with a fix/patch to the codes that were considered vulnerable. Again, you must explain in detail the steps you have taken to obtain the best response from ChatGPT. You must also discuss the reactions you have received, namely, whether they were appropriate.	34

Lista de Acrónimos

AFL *American Fuzzy Lop*
API *Application Programming Interface*
DNS *Domain Name System*
DNSSEC *Domain Name System Security Extensions*
HTML *HyperText Markup Language*
HTTP *HyperText Transfer Protocol*
OWASP *Open Web Application Security Project*
PHP *Hypertext Preprocessor*
RIP *Register Instruction Pointer*
RBP *Register Base Pointer*
RR *Resource Record*
SQL *Structured Query Language*
SQLi *SQL Injection*
XSS *Cross-Site Scripting*

Lista de Figuras

FIGURA 1		9
FIGURA 2		10
FIGURA 3		11
FIGURA 4		11
FIGURA 5		12
FIGURA 6		13
FIGURA 7		13
FIGURA 8		14
FIGURA 9		15
FIGURA 10		15
FIGURA 11		16
FIGURA 12		16
FIGURA 13		16
FIGURA 14		17
FIGURA 15		18
FIGURA 16		18
FIGURA 17		19
FIGURA 18		19
FIGURA 19		21
FIGURA 20		21
FIGURA 21		22
FIGURA 22		22
FIGURA 23		23
FIGURA 24		25
FIGURA 25		27
FIGURA 26		28
FIGURA 27		29
FIGURA 28		29
FIGURA 29		29
FIGURA 30		30
FIGURA 31		30
FIGURA 32		30
FIGURA 33	FIGURA 34	33
FIGURA 35	FIGURA 36	35

First Part

2. VulnApp

01. Characterize the attack surface of each application. Justify your answer by relating it to the code.

Um *attack surface* é a soma dos diferentes pontos de acesso por onde um atacante pode explorar vulnerabilidades na aplicação e roubar informação (*user interface, network, file system, interface w/external software, operating system e application*). Neste projeto, os diferentes pontos de ataque serão as interfaces das aplicações **map_path**, **mime**, **mysig**.

1. Relativamente ao map_path, a attack surface é caracterizada desta maneira:

- **File System:**

A aplicação lê de um ficheiro especificado pelo utilizador (`fopen(argv[1], "r")`). Como o atacante controla o conteúdo deste ficheiro, este pode mudar o caminho processado pelo programa. Isto é bastante significativo visto que dados maliciosos no ficheiro podem explorar as vulnerabilidades existente no ficheiro.

- **User Interface:**

Os argumentos passados na linha de comandos '**argc**' e '**argv**' servem como interface para o utilizador interagir com aplicação. Passar como argumento um ficheiro potencialmente malicioso irá alterar o comportamento do programa e potencialmente originar falhas de segurança.

2. Relativamente a mime:

- **File System:**

O uso das funções `fopen()` e `fgetc()` para a leitura de ficheiros faz com que o sistema de ficheiros seja uma superfície de ataque. Como o atacante controla o ficheiro de *input* (com objetivo de criar um ficheiro falsificado), podem explorar a aplicação com base no acesso de ficheiros.

- **User Interface:**

Este programa recebe um caminho de ficheiro como *input* da linha de comandos correspondente ao argumento '**argv**' (`argv[1]`). Este *input* do utilizador permite a exploração de um atacante, ao passar dados maliciosos para o ficheiro sendo assim possível executar um ataque de *path traversal*.

3. Relativamente a mysig:

- **Input Handling and Parsing:**

As funções como `RRExtract()` e `ns_nameok()` convertem e validam os *inputs* como nomes *domain* e registos DNS. A lógica de converter os nomes *domain* com `dn_expand()` e extraem vários campos de dados de um *buffer* com `GETSHORT()` e `GETLONG()`. A superfície de ataque principal relaciona-se com o *parsing the messages DNS* não confiáveis. *Packets* DNS maliciosos podem ser usados para ativar erros *parsing* ou *buffer overflows*.

02. The map path application has several vulnerabilities, including several buffer overflows. How many buffer overflow vulnerabilities do you find, and where (line number) do they occur?

Todas as vulnerabilidades identificadas na aplicação **map_path** estão relacionadas com *buffer overflows*. A primeira vulnerabilidade encontramos na **linha 29**: `strcpy(path, mapped_path)`, onde esta função faz a copia de *strings* de *mapped_path* para o *path*, incluindo o carácter '\0'. Visto que esta função não recorre a um parâmetro de verificação do tamanho do *buffer path*, esta está vulnerável a um *buffer overflow*, sobrepondo a memória alocada para outros recursos, caso o **mapped_path** seja demasiado grande.

Nas seguintes **linhas 34 e 35**, declaramos **pathspace** e **old_mapped_path** com **MAXPATHLEN**, que corresponde a 20 bytes. Apesar de serem definidas como variáveis globais para evitar *stack overflow*, ainda podem originar um *buffer overflow*, uma vez que qualquer *input* que exceda o tamanho dos *buffers* pode provocar modificações indesejadas na *stack*. Isto é particularmente evidente na função `map_dir_chdir`, onde encontramos outra vulnerabilidade, no recurso das funções `strcpy(old_mapped_path, mapped_path)` na **linha 84** e `strcpy(path, orig_path)` na **linha 87**. Adicionalmente, na **linha 113**, o `strcpy(mapped_path, old_mapped_path)` também é uma vulnerabilidade que pode resultar num *buffer overflow*.

Para explorar a vulnerabilidade do *buffer overflow*, basta ser fornecido um *input* com um caminho maior que **MAXPATHLEN**. Por exemplo, ao escrever num ficheiro de texto um caminho como **/home/ss/apps/vulnApp/map_path**, é possível desencadear o *buffer overflow*.

03. Describe an input to the program that exploits one of your identified vulnerabilities.

```
ss@ss-VirtualBox:~/apps/vulnApp/map_path$ echo -n "/home/ss/apps/vulnApp/map_path" > f.txt
ss@ss-VirtualBox:~/apps/vulnApp/map_path$ ./map_path f.txt
Current directory = /home/ss/apps/vulnApp/map_path
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
ss@ss-VirtualBox:~/apps/vulnApp/map_path$
```

Figura 1

Na Figura 1 é possível observar a execução de um comando que resulta numa mensagem de erro "**stack smashing detected**", que indica que uma vulnerabilidade de *buffer overflow* foi explorada. Neste caso, o input fornecido na aplicação **map_path** é um ficheiro de texto vazio (**f.txt**), criado com o comando:

```
echo -n "" > /home/ss/apps/vulnApp/map_path/f.txt
```

Foi preciso adicionar a *flag -n* para remover a *new line* acrescentada no final do ficheiro de texto. O programa é então executado com o seguinte comando, onde foi usado o ficheiro criado que possui um caminho maior do que o esperado pela aplicação para causar um *buffer overflow*:

```
./map_path f.txt
```

Esta vulnerabilidade de *buffer overflow* é explorada quando o programa processa o *input* do ficheiro '**f.txt**', e é identificado que existe uma falha na gestão dos tamanhos dos *buffers* internos. Se o programa não verifica corretamente o tamanho da entrada, o *buffer* pode ser sobrescrito, ocorrendo assim uma corrupção de memória, tal como demonstrado na figura, com a mensagem do sistema "**stack smashing detected**", resultando num crash ou potencial comprometimento da segurança.

04. The mime application also has several buffer overflow vulnerabilities, this time in the heap and related to pointer manipulation. Please find one of the vulnerabilities and explain why it is exploitable.

A aplicação **mime** apresenta uma vulnerabilidade de *buffer overflow* no **heap** (local da memória de alocação dinâmica, ou seja, *pointers*) com recurso à manipulação de apontadores, durante o processamento do *buffer fbuf* na função **MIME_func**.

A vulnerabilidade **heap** na função provém de verificações de limites incorretos e manipulação de *pointers* quando mexemos com o **fbuf** *buffer* e no *pointer fbufp*. O problema está na maneira de como o **fbufp** é usada para escrever data no *buffer fbuf*, sendo que há exemplos no código em

que não há uma verificação correta, ou de todo, do limite deste buffer:

```
136     c2 = CHAR64(c2);
137
138     /*We have read in four characters */
139     *fbufp = (c1 << 2) | ((c2 & 0x30) >> 4); /* Write first char to fbuf */
140
141     /* Check to see if either 1) reached a '\n' or */
142     /* 2) fbufp about to point past end of fbuf */
143
144     if (*fbufp++ == '\n' || fbufp >= &fbuf[MAXLINE])
145     {
146         if (*--fbufp != '\n' || *--fbufp != '\n')
147             fbufp++;
148
149         *fbufp = '\0';
150         /* putline((char *) fbuf, mci); */
151         printf("resetting fbufp\n");
152         fbufp = fbuf;
153     }
154
155     if (c3 == '-')
156         continue;
157     c3 = CHAR64(c3);
158
159     *fbufp = ((c2 & 0x0f) << 4) | ((c3 & 0x3c) >> 2);
160
161     if (*fbufp++ == '\n' || fbufp >= &fbuf[MAXLINE])
162     {
163
164         if (*--fbufp != '\n' || *--fbufp != '\n')
165             fbufp++;
166         *fbufp = '\0';
167         /* putline((char *) fbuf, mci); */
168         printf("resetting fbufp\n");
169         fbufp = fbuf;
170     }
171 }
```

Figura 2

Como se pode observar na Figura 2, na **linha 144** do programa temos a comparação incorreta do `fbufp >= &fbuf[MAXLINE]`, devia comparar o **fbufp** com o `&fbuf[MAXLINE]` em vez do `fbuf`. Também se faz o incremento do **fbufp** antes de verificar se está nos limites do buffer `fbuf` que pode causar escritas fora desses limites caso por exemplo o **fbufp** apontar para `&fbuf[MAXLINE-1]`, saindo para fora do buffer alocado. Tal acontece para o *underflow* na **linha 165**, no caso do **fbufp** já estiver no limite mais baixo e decrementarmos na mesma.

Demonstração de dois exemplos, um funcional e um não funcional (foi adicionado um **printf** para demonstrar o valor decodificado). No primeiro exemplo estamos a converter uma base 64 `"QUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVoQUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVo="`, que está dentro dos limites do `fbuf`.

No segundo caso:

`"QUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVoQUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVo="`,

Na Figura 3 é possível observar que, como o comprimento é maior que o **MAXLINE** definido, resulta numa decodificação incorreta dos valores, o que faz sentido sendo que claramente

mostra um **heap overflow** em que a corrupção de memória ocorre, e maus caracteres aparecem no output.

```
ss@ss-VirtualBox:~/apps/vulnApp/mime$ nano f.txt
ss@ss-VirtualBox:~/apps/vulnApp/mime$ ./mime f.txt
Decoded value: ABCDEFGHIJKLMNOPQRSTUVWXYZ
ss@ss-VirtualBox:~/apps/vulnApp/mime$ nano f.txt
ss@ss-VirtualBox:~/apps/vulnApp/mime$ ./mime f.txt
Decoded value: ABCDEFGHIJKLMNOPQRSTUVWXYZ
ss@ss-VirtualBox:~/apps/vulnApp/mime$
```

Figura 3

05. How should the program be modified to prevent this vulnerability? Explain how your patch is effective at preventing the exploitation of the flaw.

Para evitar a vulnerabilidade **heap overflow**, podemos modificar o código com melhores verificações de limites, melhor validação de input e uma gestão de buffer mais segura:

Primeiramente decidiu-se criar uma variável **bytes_written** para no final verificar se não excedeu o **MAXLINE**, ou seja, se não conseguiu escrever mais *bytes* no *buffer* dos que eram possíveis.

Depois desta criação da variável, em cada leitura que se fazia do **fbufp**, onde se escrevia cada caractere para o **fbuf**, incrementava-se o pointer e a variável **bytes_written**:

```

138
139      /*We have read in four characters */
140      *fbufp = (c1 << 2) | ((c2 & 0x30) >> 4); /* Write first char to fbuf */
141      fbufp++;
142      bytes_written++;
143
144      /* Check to see if either 1) reached a '\n' or */
145      /* 2) fbufp about to point past end of fbuf */
146
147      if (fbufp >= &fbuf[MAXLINE])
148      {
149
150          *fbufp = '\0';
151          /* putline((char *) fbuf, mci); */
152          printf("resetting fbufp\n");
153          fbufp = fbuf;
154
155      }
156

```

Figura 4

Na Figura 4 é possível observar que, isto fez com que em cada iteração e escrita do *buffer* que se fazia, verificava-se sempre se era possível continuar a preencher o *buffer* ou se teríamos de encerrar o processo porque já tínhamos chegado ao seu fim.

Na Figura 5 é demonstrado o resultado da implementação da solução, com o mesmo *input* unsafe que tínhamos dado para testar a vulnerabilidade.

```
diogo@LAPTOP-DR3EQNEV:/mnt/c/Users/diogo/Desktop/DI/Mestrado/1º Ano/1º Semestre/Segurança de Software/Projeto
$ echo "QUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVoQUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVo=" > f.txt
diogo@LAPTOP-DR3EQNEV:/mnt/c/Users/diogo/Desktop/DI/Mestrado/1º Ano/1º Semestre/Segurança de Software/Projeto
$ ./mime f.txt
Error: File size (72) exceeds MAXLINE limit 50.
diogo@LAPTOP-DR3EQNEV:/mnt/c/Users/diogo/Desktop/DI/Mestrado/1º Ano/1º Semestre/Segurança de Software/Projeto
$
```

Figura 5

06. The *mysig* application may have an integer overflow flaw that allows for a buffer overflow. Could you confirm if this problem exists? Explain how the adversary could exploit the bug.

Confirmamos que a aplicação **mysig** tem uma vulnerabilidade de **integer overflow**, ou seja, excede o valor máximo permitido (vulnerabilidade de inteiros mais comum). Pode levar a comportamentos inesperados da aplicação, como bypass de verificações de limites. Esta vulnerabilidade ocorre na função `createSig`, que cria uma **signature record** num *buffer bf*, onde o comprimento do *buffer* é controlado pela variável **len**. Durante a construção da assinatura, **len** é incrementado ao longo do processo com recurso do **comp_size**, sem uma verificação de limite rigorosa. Se o valor de **len** ultrapassar o tamanho máximo permitido, ele pode resultar em um **integer overflow** voltando a valores baixos ou negativos. Assim, **len** pode indicar que o *buffer* ainda tem espaço disponível, quando na realidade este já foi excedido.

Um atacante poderia explorar esta vulnerabilidade fornecendo um **comp_dn** que faça o valor de **len** ultrapassar o limite permitido (ou que o aproxime deste limite), conseguindo escrever além dos limites do *buffer*, manipulando a aplicação para escrever em memória adjacente.

Existe também um **integer overflow** na função `newstr`, esta função que aloca um *buffer* com base no tamanho do parâmetro **len**, mas há uma condição que leva a um **integer overflow**.

```

* Return a counted string buffer big enough for a string of length 'len'.
*/
char *
newstr(size_t len, int needpanic) {
    u_char *buf, *bp;

    assert(len <= 65536);

    buf = (u_char *)malloc(2 + len + 1);
    if (buf == NULL) {
        if (needpanic)
            printf("savestr: malloc failed (%s)", strerror(errno));
        else
            return (NULL);
    }
    bp = buf;
    NS_PUT16(len, bp);
    return ((char *)bp);
}

```

Figura 6

Como se pode observar na Figura 6, a verificação observa se “**len <= 65536**”, permitindo assim que reside nesse limite, mas a alocação do *buffer* envolve adicionar valores (**2 + len + 1**), o que pode exceder o valor máximo de **size_t**, potencialmente resultando no **overflow** ou **underflow**, em que se transformam em valores negativos.

07. Explain how you would modify the application to prevent the flaw’s exploitation.

Na Figura 7, mostra-se que uma possível verificação que a conta aritmética não excederá o limite do **len**, causando um **integer overflow**. Isto permite encerrar o programa antes de haver este overflow.

```

if(len > SIZE_MAX - 3) {
    if(needpanic)
        printf("savestr: size exceeds allowable limit\n");
    return NULL;
}

```

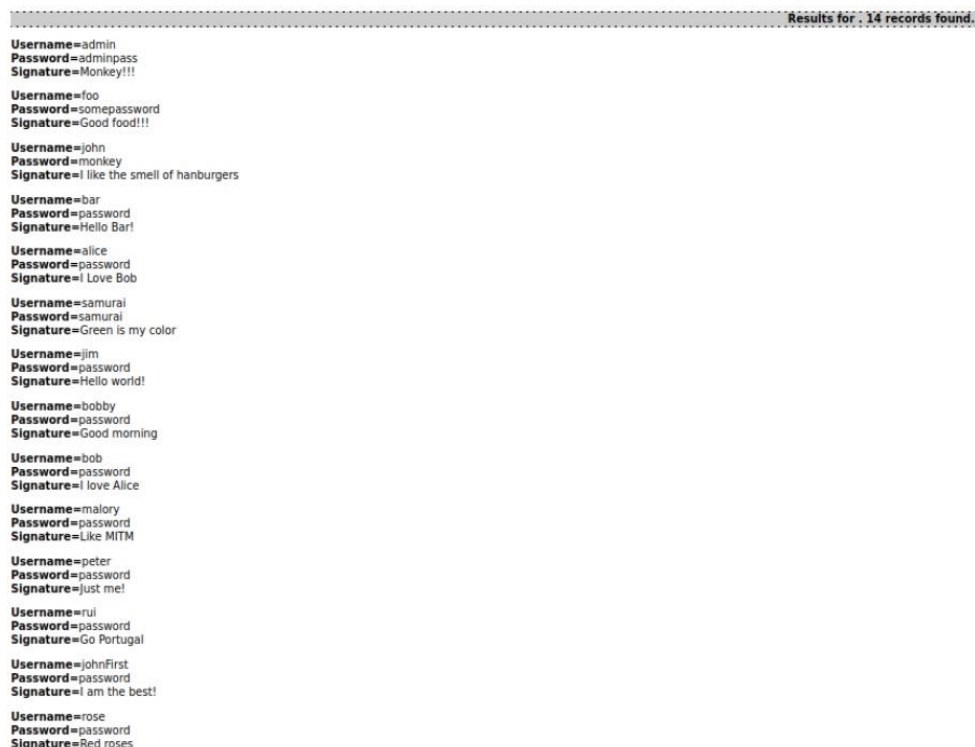
Figura 7

3. SSS-DB

08. The Show User Info option of Operations is vulnerable to a SQL injection attack (file user-info.php). Perform this attack and get a listing of all the users and passwords of the application. Look at the file user-info.php and explain why this

happens.

Recorreu-se ao uso do conceito chamado tautologias, que neste caso são condições que são sempre verdadeiras, onde se injetou o seguinte comando SQL: `' OR 1=1 --`. Como nós não tínhamos conhecimento de nenhum *username* possível começamos por fechar esse campo com o caracter `'` e de seguida utilizamos uma tautologia `1 = 1`. Assim, como esta condição é sempre verdadeira, ao utilizar a lógica sabemos que algo verdadeiro ou [qualquer coisa] será sempre verdade. Acabamos ainda o nosso input com os caracteres `--` que em SQL têm o significado de comentar tudo o que se encontra à sua frente. Dado isto, com base na resposta do serviço podemos confirmar que o campo está vulnerável a **SQL Injection**.



```
Results for : 14 records found.

Username=admin
Password=adminpass
Signature=Monkey!!!

Username=foo
Password=somepassword
Signature=Good food!!!

Username=john
Password=monkey
Signature=I like the smell of hanburgers

Username=bar
Password=password
Signature=Hello Bar!

Username=alice
Password=password
Signature=I Love Bob

Username=samurai
Password=samurai
Signature=Green is my color

Username=jim
Password=password
Signature=Hello world!

Username=bobby
Password=password
Signature=Good morning

Username=bob
Password=password
Signature=I love Alice

Username=malory
Password=password
Signature=Like MITM

Username=peter
Password=password
Signature=just me!

Username=rui
Password=password
Signature=Go Portugal

Username=johnFirst
Password=password
Signature=I am the best!

Username=rose
Password=password
Signature=Red roses
```

Figura 8

A Figura 8 é possível verificar que ocorreu com sucesso a execução do **SQL Injection** com a injeção do seguinte query: `tom' UNION select * from accounts --`. Esta query sql permite visualizar os dados da tabela de utilizadores e senhas da aplicação, tal como pretendido. Neste caso, começamos por supor um possível utilizador, “**tom**”, e reunimos esta *query* com a seguinte *query*: `SELECT * FROM accounts`, com o objetivo de selecionar todos os campos da tabela “**accounts**”, permitindo assim termos acesso a toda essa tabela. Ao utilizar a *query* de **UNION**, sabemos que ambos os **selects** vão ser feitos em vez de apenas um. Isto é possível porque se trata de *queries* **SELECT** e o domínio e campos a extrair da tabela são os mesmos. A vulnerabilidade existe porque o input do utilizador não é devidamente **sanitizado** antes de ser utilizado em consultas SQL.

09. The Show Log option of the Core Controls menu entry allows a stored XSS attack. Explain why and describe how the attack could be performed (the attack can show a popup window saying “hello”).

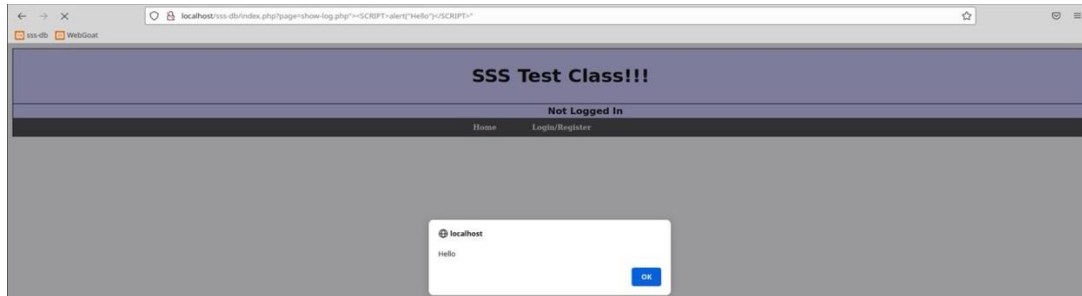


Figura 9

Na Figura 9 é possível ver um *pop-up* com a mensagem “**hello**”, demonstrando a execução de um código malicioso injetado na aplicação. Esta ação demonstra claramente o resultado de um ataque **stored XSS**, onde um script malicioso foi injetado na *tab* “**Show User Info**”, na aplicação, como utilizador e password, e exibido quando a página de log for acedida. Isto confirma que a aplicação não trata adequadamente entradas de utilizadores, permitindo a injeção de scripts.

10. The option Text File Viewer of the Operations menu allows the display of a specific document about hacking (file text-file-viewer.php). This option is vulnerable to a reflected XSS attack even though the user does not write any text, and they can only choose a document from a fixed list. Explore this vulnerability to show, for example, a popup window saying “hello.”



Figura 10

Na Figura 10 observamos o uso da ferramenta **OWASP ZAP** para intercepar e modificar uma requisição **POST** da aplicação. O campo **textfile** foi alterado para o código malicioso `<script>alert('hello')</script>`, demonstrando a exploração de uma vulnerabilidade de **Reflected XSS**.

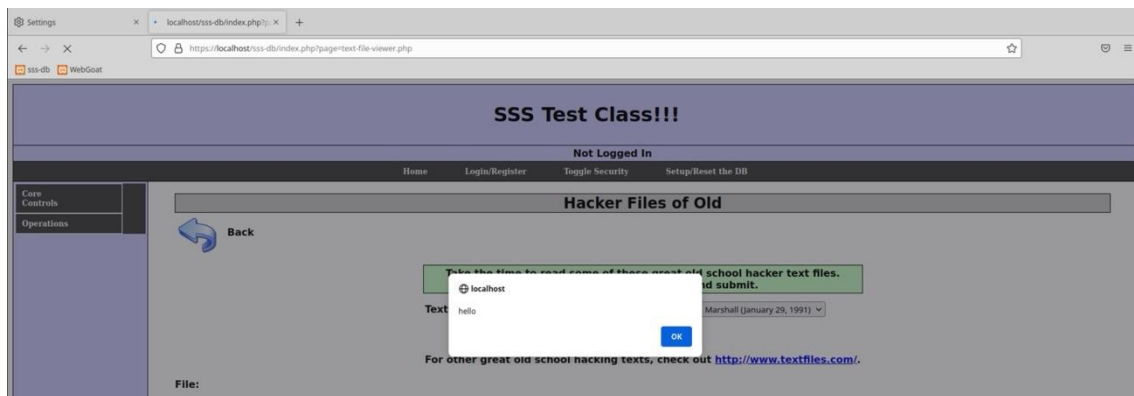


Figura 11

Na Figura 11 é possível observar o resultado da execução do ataque **XSS**. A página da aplicação exibe um *pop-up* com a mensagem "**hello**", confirmando que o código injetado foi executado com sucesso no navegador.

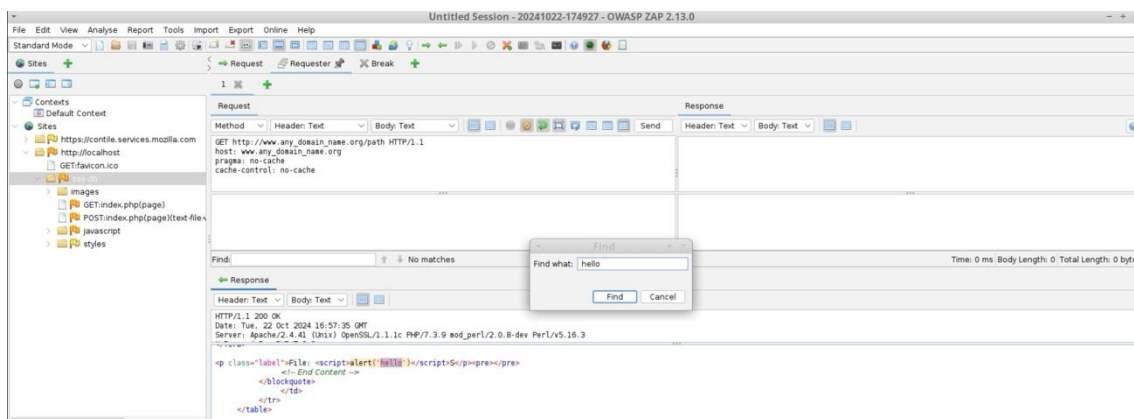


Figura 12

Na Figura 12 podemos ver a resposta da aplicação, capturada pelo **ZAP**. O código HTML retornado contém o script malicioso injetado, confirmando que a aplicação refletiu o conteúdo sem qualquer validação ou sanitização.

11. Explain why this vulnerability exists by looking into text-file-viewer.php. How could you protect yourself from this attack?

```
try {
    switch ($_SESSION["security-level"]){
        case "0":
        case "1":
            $lUseTokenization = FALSE;
            break;

        case "2":
        case "3":
        case "4":
        case "5":
            $lUseTokenization = TRUE;
            break;
    }// end switch ($_SESSION["security-level"])
```

Figura 13

Na Figura 13 é possível observar o uso de uma estrutura `switch` no código que configura o nível

de segurança da aplicação com base na sessão do utilizador (`$_SESSION["security-level"]`). No **nível de segurança 0, 2, 3 e 4** verificamos que não existem nenhuma proteção contra ataques XSS. No **nível de segurança 1**, a tokenização está desativada (`$UseTokenization = FALSE`), onde a tokenização é um processo de segurança que permite a encriptação de inputs em *tokens*, logo se esta variável não está ativa, quer dizer que não estamos protegidos com a ajuda dela. Enquanto no nível de segurança 5, a tokenização já se encontra ativada (`$UseTokenization = TRUE`), logo podemos usufruir dos processos de segurança de tokenização, que servem para validar o input.

Podemos afirmar que não existe validação de input sem ser no caso de **nível de segurança 5**, onde é possível injetar um script malicioso através do campo *textfile* para explorar a vulnerabilidade de **XSS**.

```
elseif ($UseTokenization){
    $pTextFile=$_POST["textfile"];

    $isDigits = (preg_match("/\d{1,2}/", $pTextFile) == 1);
    if ($isDigits && $pTextFile > 0 && $pTextFile < 11){
        $URL = "";
        switch($pTextFile){
            case 1: $URL = "./FILES/auditoool.txt";break;
            case 2: $URL = "./FILES/atms";break;
            case 3: $URL = "./FILES/backdoor.txt";break;
            case 4: $URL = "./FILES/hack1.hac";break;
            case 5: $URL = "./FILES/hacking101.hac";break;
        }
        // end switch($pTextFile)

        $LogHandler->writeToLog($conn, "Displayed contents of URL: " . $URL);

        try{
            // open file handle
            $handle = fopen($URL, "r");
            echo '<p class="label">File: '.$URL.'</p>';
            echo '<pre>';
            echo stream_get_contents($handle);
            echo '</pre>';
            fclose($handle);
        }catch(Exception $e){
            echo $CustomErrorHandler->FormatError($e, "Error opening file stream. Cannot load file.");
        }
        // end try
    }
}
```

Figura 14

Na Figura 14 é possível observar a parte do código que valida o input do campo **textfile** quando a **tokenização** está ativada. A entrada é verificada usando uma expressão regular para garantir que o valor seja um número entre 1 e 10. Dependendo do valor, um arquivo específico é selecionado para leitura e exibição ao utilizador.

12. The DNS Lookup option of the Operations menu entry is vulnerable to a command injection vulnerability (file dns-lookup.php). Explore this vulnerability to get the credentials of access to the application's database (host, database name, username and password).

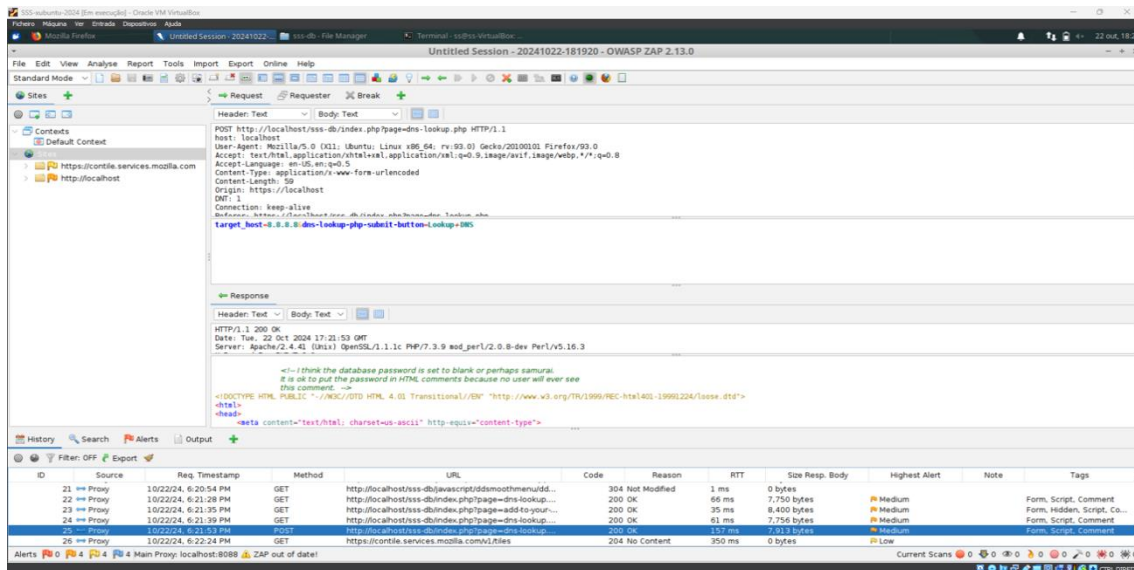


Figura 15

A Figura 15 demonstra o uso do **OWASP ZAP** para manipular a requisição enviada para a aplicação. O campo **target_host** foi alterado para incluir o seguinte comando malicioso:

8.8.8.8; ls

Esse comando tenta explorar uma vulnerabilidade de injeção de comandos no parâmetro **target_host**, permitindo a execução do comando **ls** para exibir o conteúdo do diretório



Figura 16

Na Figura 16 é possível observar que a aplicação responde com a lista de vários arquivos e *scripts* PHP disponíveis no servidor. Isso indica que o comando injetado foi executado com sucesso e que o conteúdo do diretório foi revelado. No entanto, o objetivo é procurar informações mais específicas, como as credenciais do base de dados.

onde a aplicação implementa uma validação adequada por meio de expressões regulares e sanitização de input.

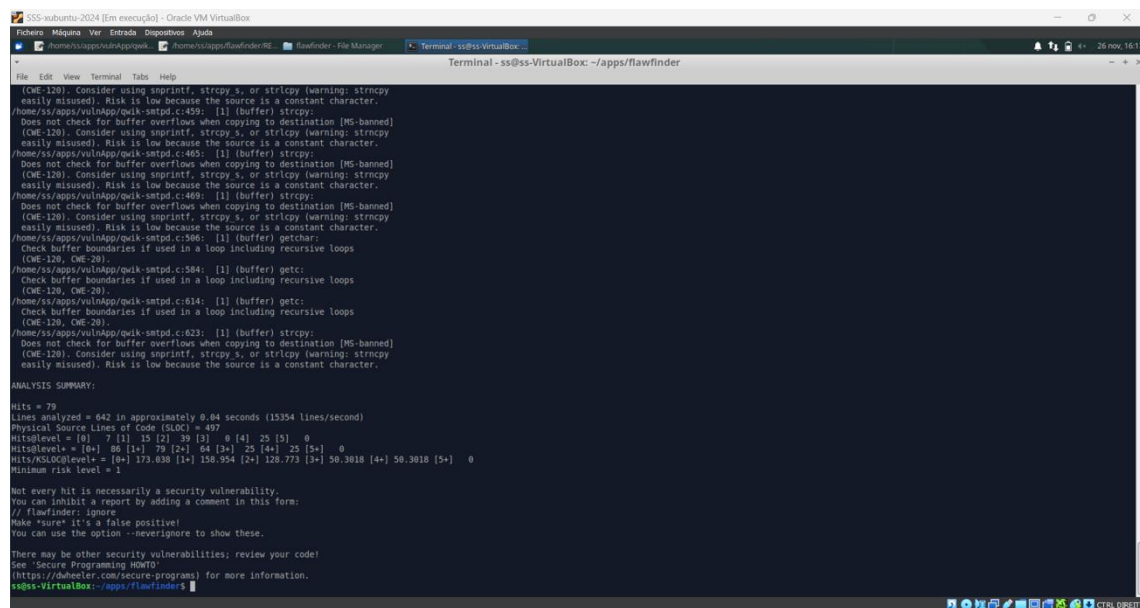
- **Nível de Segurança 5:** A entrada é validada com expressões regulares (`preg_match`) para garantir que o valor seja um endereço IPv4, IPv6, ou um nome de domínio válido. Além disso, o valor de `target_host` é sanitizado com a função `$encoder->encodeForHTML()`, o que evita a execução de comandos ou scripts maliciosos. Esta técnica irá prevenir de ataques que contenham metacharacters, que foi algo de que nos aproveitamos para explorar a vulnerabilidade. Este caso ainda utiliza outros mecanismos de proteção como fazer um teste de codificação de html da variável `$ITargetHostText` e ativar as validações de javascript, `$IEnableJavaScriptValidation`.

Para a proteção da aplicação contra vulnerabilidades deste tipo seria necessário aplicar mecanismos de segurança como no caso de **nível de segurança 5**, onde se validam os inputs conforme uma expressão regular (como uma *whitelist*) e validam os campos de html com mecanismos de validação do javascript.

Second Part

4. Flawfinder

14. At the folder of the tool (/home/ss/apps/flawfinder), execute the next command: `./flawfinder -m 2 ../vulnApp/mysig/*` . For each error reported by the tool, indicate if it is actually a vulnerability.



```
File Edit View Terminal Tabs Help
/home/ss/apps/vulnApp/mysig/... /home/ss/apps/flawfinder/RE /flawfinder - File Manager
Terminal - ss@ss-VirtualBox: ~/apps/flawfinder

CWE-120) Consider using snprintf, strcpy_s, or strncpy (warning: strcpy
easily misused). Risk is low because the source is a constant character.
/home/ss/apps/vulnApp/qwik-smtpd.c:459: [1] (buffer) strcpy:
Does not check for buffer overflows when copying to destination [MS-banned]
CWE-129) Consider using snprintf, strcpy_s, or strncpy (warning: strcpy
easily misused). Risk is low because the source is a constant character.
/home/ss/apps/vulnApp/qwik-smtpd.c:465: [1] (buffer) strcpy:
Does not check for buffer overflows when copying to destination [MS-banned]
CWE-128) Consider using snprintf, strcpy_s, or strncpy (warning: strcpy
easily misused). Risk is low because the source is a constant character.
/home/ss/apps/vulnApp/qwik-smtpd.c:469: [1] (buffer) strcpy:
Does not check for buffer overflows when copying to destination [MS-banned]
CWE-129) Consider using snprintf, strcpy_s, or strncpy (warning: strcpy
easily misused). Risk is low because the source is a constant character.
/home/ss/apps/vulnApp/qwik-smtpd.c:586: [1] (buffer) getc:
Check buffer boundaries if used in a loop including recursive loops
CWE-120, CWE-20).
/home/ss/apps/vulnApp/qwik-smtpd.c:584: [1] (buffer) getc:
Check buffer boundaries if used in a loop including recursive loops
CWE-120, CWE-20).
/home/ss/apps/vulnApp/qwik-smtpd.c:614: [1] (buffer) getc:
Check buffer boundaries if used in a loop including recursive loops
CWE-120, CWE-20).
/home/ss/apps/vulnApp/qwik-smtpd.c:623: [1] (buffer) strcpy:
Does not check for buffer overflows when copying to destination [MS-banned]
CWE-120) Consider using snprintf, strcpy_s, or strncpy (warning: strcpy
easily misused). Risk is low because the source is a constant character.

ANALYSIS SUMMARY:
Hits = 79
Lines analyzed = 642 in approximately 0.04 seconds (15354 lines/second)
Physical Source Lines of Code (SLOC) = 487
Hits@Level = [0] 7 [1] 15 [2] 39 [3] 0 [4] 25 [5] 0
Hits@Level+ = [0+] 86 [1+] 79 [2+] 64 [3+] 25 [4+] 25 [5+] 0
Hits@SLOCLevel+ = [0+] 173.030 [1+] 150.954 [2+] 128.773 [3+] 50.3018 [4+] 50.3018 [5+] 0
Minimum risk level = 1

Not every hit is necessarily a security vulnerability.
You can inhibit a report by adding a comment in this form:
// flawfinder: ignore
Make "sure" it's a false positive!
You can use the option -neverignore to show these.

There may be other security vulnerabilities; review your code!
See 'Secure Programming HOWTO'
(https://dhwheeler.com/secure-programs) for more information.
ss@ss-VirtualBox: ~/apps/flawfinder
```

Figura 19

Na Figura 19 é possível observar o resultado devolvido pela ferramenta **FlawFinder**, que fez análise estática da aplicação **mysig**. A ferramenta identificou **79 Hits** e uma classificação na análise um **nível de risco mínimo igual a 1**.



```
307 /* loop off the TTL at the expiration time. */
308 timetilexp = exptime - now;
309 if (timetilexp < ttl) {
310     printf("shrinking expiring SIG TTL");
311     ttl = timetilexp;
312 }
313
314 cp = cpl - 10;
315
316 cpl = (u_char *)data;
317
318 /* first just copy over the type covered, algorithm, */
319 /* labels, orig ttl, two timestamps, and the footprint */
320 BOUNDS_CHECK(cp, 10);
321 memcpy(cp, cp, 10);
322
323 cpl += 10;
324
325 /* then the signer's name */
326 printf("sizeof data = %d\n", sizeof data);
327 printf("comp name = %s\n", (char *) cp+10);
328
329 n = dn_expand(msg, eom, cp+10, (char *)cpl, (sizeof data) - 10);
330
331 /* n is the length of the compressed domain name as seen in msg */
332 printf("dn_expand returned: %d, expanded name = %s\n", n, (char *)cpl);
333
334 if (n < 0) {
335     printf("ERROR: n = %d < 0\n", n);
336     printf("EXITING RREXTRACT!\n");
337     rc =code = FORMERR;
338     return (-1);
339 }
340 }
```

Figura 20

Na Figura 20, a **linha 402** contém uma chamada para a função `memcpy`, que está a ser usada para copiar 18 bytes de dados entre os buffers "**cp**" e "**cp1**". Contudo, ao contrário do que possa inicialmente ser assumido, não existe uma potencial vulnerabilidade de segurança neste uso específico da função `memcpy`. Isto deve-se ao facto que na linha imediatamente anterior recorre-se à função `BOUNDS_CHECK`, esta que assegura o tamanho do *buffer* de destino, sendo assim validado de forma apropriada, não ocorrendo uma possível vulnerabilidade de *buffer overflow*.

```

400     if (n > (int)(sizeof data) - (cp1 - (u_char *)data)) {
401         printf("NO ROOM\n");
402         hp->rcode = FORMERR;
403         return (-1); /* out of room */
404     }
405
406     switch (alg) {
407     case NS_ALG_MD5RSA:
408         if (n < NS_MD5RSA_MIN_SIZE || n > NS_MD5RSA_MAX_SIZE)
409             break;
410         hp->rcode = FORMERR;
411         break;
412     case NS_ALG_DSA:
413         if (n != NS_DSA_SIG_SIZE)
414             break;
415         hp->rcode = FORMERR;
416         break;
417     default:
418         printf("DEFAULT ALG\n");
419         break;
420     }
421
422     if (hp->rcode == FORMERR)
423         return (-1);
424
425     memcpy(cp1, cp, n);
426     cp += n;
427     cp1 += n;
428
429     /* compute size of data */
430     n = cp1 - (u_char *)data;
431     cp1 = (u_char *)data;
432     break;
433 }
434
435 default:
436     printf("unknown type %d", type);
437     return ((cp - rrp) + dien);
438 }

```

Figura 21

Na Figura 21 é possível observar demonstra um trecho onde a função `memcpy` é usada na **linha 456** para copiar dados de um *buffer* **cp** para outro *buffer* **cp1**. A quantidade de dados copiados é definida pela variável "**n**" e o código não realiza uma verificação explícita do tamanho do *buffer* de destino antes da cópia. Isso corresponde à vulnerabilidade que alerta para o uso inseguro de `memcpy` sem verificações adequadas, que pode ser substituída por `memmove` para mitigar riscos de segurança como o *buffer overflow*.

```

500 /* create the signature file this model needs */
501
502 int createsig (u_char *buf) {
503     u_char *p;
504     char *temp;
505     u_char *comp_dn, *comp_dn2;
506     [char exp_dn[200], exp_dn2[200]];
507     u_char *dnptrs, *lastdnptr, *dnptrs2;
508     int i, len = 0, comp_size;
509     u_long now;
510
511     dnptrs = (unsigned char *) malloc(1 * sizeof(unsigned char *));
512     dnptrs2 = (unsigned char *) malloc(1 * sizeof(unsigned char *));
513     comp_dn = (unsigned char *) malloc(200 * sizeof(unsigned char));
514     comp_dn2 = (unsigned char *) malloc(200 * sizeof(unsigned char));
515     temp1 = (char *) malloc(400 * sizeof(char));
516
517     temp = temp1;
518     p = buf;
519
520     strcpy(temp, "HEADER JUNK:");
521     len += strlen(temp);
522     while (*temp != '\0')
523         *p++ = *temp++;
524     strcpy(exp_dn, "ics.mit.edu");
525     *dnptrs++ = (u_char *) exp_dn;
526     *dnptrs-- = NULL;
527     lastdnptr = NULL;
528     printf("calling dn.comp.\n");
529     comp_size = dn.comp((const char *) exp_dn, comp_dn, 200, dnptrs, lastdnptr);
530     printf("comp size = %d\n", strlen(exp_dn));
531     printf("comp size = %d\n", comp_size);
532     printf("exp_dn = %s, comp_dn = %s\n", exp_dn, (char *) comp_dn);

```

Figura 22

Na Figura 22 observa-se a função `strcpy` sendo utilizada na **linha 524** para copiar a *string* "HEADER JUNK" para o *buffer* **temp**. Contudo, ao contrário do que poderia ser inicialmente identificado como um *buffer overflow*, neste caso não existe, pois a *string* copiada é **hardcoded** de tamanho fixo e o *buffer* de destino **temp** tem 400 bytes disponíveis, garantindo que haja espaço suficiente para armazenar a *string* sem risco de *buffer overflow*. Na **linha 531**, a função

`strcpy` é utilizada para copiar a *string* hardcoded "**lcs.mit.edu**" para o buffer **exp_dn**. Apesar de a função `strcpy` não verificar o tamanho do *buffer* de destino, nesta situação não existe uma vulnerabilidade, pois a *string* "**lcs.mit.edu**" tem tamanho fixo conhecido (12 caracteres, incluindo o "\0"), e o *buffer* **exp_dn** é declarado com um tamanho de 200 bytes, o que é significativamente maior do que o necessário para armazenar a *string* copiada. Logo, em ambas as **linhas 524 e 531**, as condições garantem que não ocorra risco de *buffer overflow*. Embora a função `strcpy` possa ser suscetível a vulnerabilidades em outros contextos, o uso nesta situação é seguro devido à garantia de espaço suficiente no *buffer* de destino.

```
520 PUTSHORT(30, p); /* also = len of everything starting with the covered byte (the length
521 of the entire resource record... we lie about it
522 */
523 p += 2;
524 len += 10;
525 PUTSHORT(15, p); /* covered type */
526 p += 2;
527 PUTSHORT(250*, p); /* algorithm and labels... MAKE ALG = 2, i.e default ALG */
528 p += 2;
529 PUTLONG(25, p); /* orig ttl */
530 p += 4;
531 now = time(NULL);
532 printf("Signing at = %d\n", now);
533 PUTLONG(now+20000, p); /* expiration time */
534 p += 4;
535 PUTLONG(now, p); /* time signed */
536 p += 4;
537 PUTSHORT(100, p); /* random key footprint */
538 p += 2;
539 len += 10;
540 strcpy(exp_dn2, "ccs.ocs.fcul.pt"); /* signer */
541 *dptrs2++ = (u_char *) exp_dn2;
542 *dptrs2-- = NULL;
543 lastdptr = NULL;
```

Figura 23

Na Figura 23 a função `strcpy` é utilizada na **linha 587** para copiar a *string* constante "**ccs.ocs.fcul.pt**" para o *buffer* **exp_dn2**. Neste contexto específico, o uso de `strcpy` apresenta baixo risco, já que a *string* copiada é constante e de tamanho conhecido, o que não representaria uma ameaça de segurança significativa.

Na Tabela 1.1 encontra-se todos os casos referidos anteriormente para cada linha, onde se apresenta o tipo, classificação e justificação.

Tabela 1

Linha	Tipo	Classificação
402	memcpy	Falso Positivo
456	memcpy	Verdadeiro Positivo
524	strcpy	Falso Positivo
531	strcpy	Falso Positivo
587	strcpy	Falso

		Positivo
--	--	----------

1. Substituição de memcpy:

A função `memcpy` pode ser problemática se o tamanho do destino não for cuidadosamente verificado.

- **Solução:** Utilizar `memmove`, que lida com situações de sobreposição de memória e é mais seguro. Adicionalmente, é essencial garantir que o tamanho de origem seja sempre menor ou igual ao tamanho do destino.

```
if (dest_size >= src_size) {
    memmove(destination, source, src_size);
}
```

2. Substituição de strcpy:

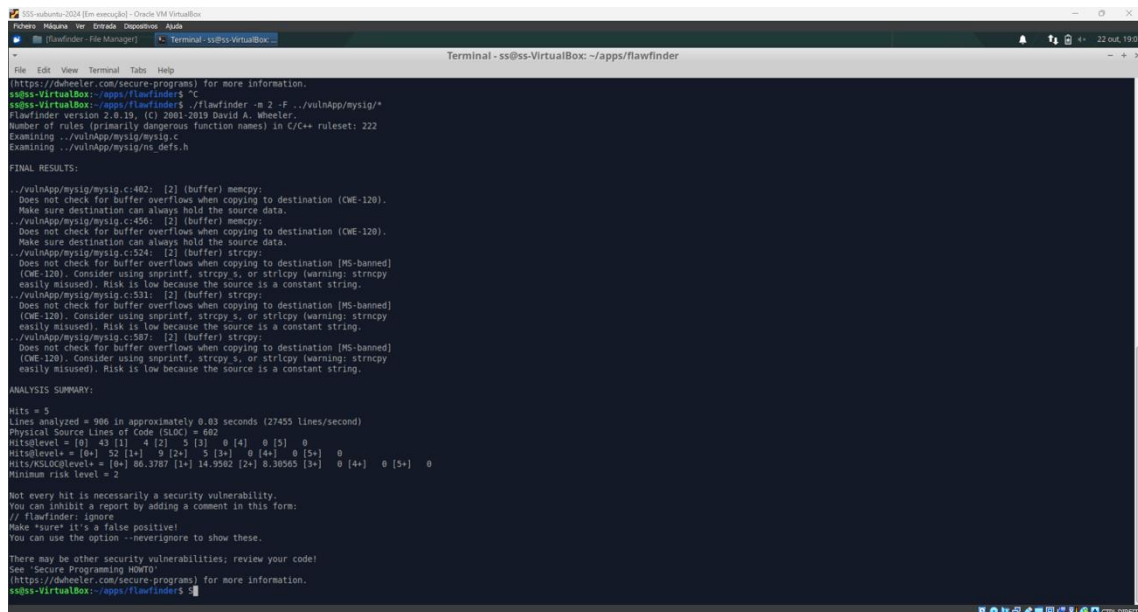
O uso da função `strcpy` sem verificação do tamanho é arriscado. A função `strncpy` ou `snprintf` são alternativas seguras porque permitem especificar o número máximo de caracteres a serem copiados.

- **Solução:**

```
strncpy(destination, source, dest_size - 1);
destination[dest_size - 1] = '\0';
```


15. Execute the same command but add the `-F` option (does not output False Positives) `./flawfinder -m 2 -F ../vulnApp/mysig/*` .

Compare the results with the analysis you did in the previous question. Is the tool precise in its results? Justify your analysis and answer in the report.



```
ss@ss-VirtualBox: ~/apps/flawfinder
$ ./flawfinder -m 2 -F ../vulnApp/mysig/*
Flawfinder version 2.0.19, (C) 2001-2019 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 222
Examining ../vulnApp/mysig/mysig.c
Examining ../vulnApp/mysig/ms_defs.h

FINAL RESULTS:

../vulnApp/mysig/mysig.c:402: [2] (buffer) memcpy:
Does not check for buffer overflows when copying to destination (CWE-120).
Make sure destination can always hold the source data.
../vulnApp/mysig/mysig.c:450: [2] (buffer) memcpy:
Does not check for buffer overflows when copying to destination (CWE-120).
Make sure destination can always hold the source data.
../vulnApp/mysig/mysig.c:524: [2] (buffer) strcpy:
Does not check for buffer overflows when copying to destination [MS-banned]
(CWE-120). Consider using snprintf, strncpy, or strlcpy (warning: strncpy
easily misused). Risk is low because the source is a constant string.
../vulnApp/mysig/mysig.c:531: [2] (buffer) strcpy:
Does not check for buffer overflows when copying to destination [MS-banned]
(CWE-120). Consider using snprintf, strncpy, or strlcpy (warning: strncpy
easily misused). Risk is low because the source is a constant string.
../vulnApp/mysig/mysig.c:587: [2] (buffer) strcpy:
Does not check for buffer overflows when copying to destination [MS-banned]
(CWE-120). Consider using snprintf, strncpy, or strlcpy (warning: strncpy
easily misused). Risk is low because the source is a constant string.

ANALYSIS SUMMARY:

Hits = 5
Lines analyzed = 986 in approximately 0.03 seconds (27455 lines/second)
Physical Source Lines of Code (SLOC) = 682
HitLevel = [0] 0 [1] 0 [2] 5 [3] 0 [4] 0 [5] 0
HitSLOCLevel = [0+] 52 [1+] 5 [2+] 5 [3+] 0 [4+] 0 [5+] 0
Hits/KSLOCLevel = [0+] 86.3787 [1+] 14.9582 [2+] 8.38565 [3+] 0 [4+] 0 [5+] 0
Minimum risk level = 2

Not every hit is necessarily a security vulnerability.
You can inhibit a report by adding a comment in this form:
// flawfinder: ignore
Make "sure" it's a false positive!
You can use the option --neverignore to show these.

There may be other security vulnerabilities; review your code!
See "Secure Programming HOWTO"
(https://dewheeler.com/secure-programs) for more information.
ss@ss-VirtualBox: ~/apps/flawfinder $
```

Figura 24

A Figura 24 confirma que a execução do **Flawfinder** com a opção `-F` resulta em uma lista mais precisa e filtrada de vulnerabilidades. Os falsos positivos foram eliminados, tornando mais fácil para os desenvolvedores concentrarem-se nos problemas reais de segurança, como o uso de funções que manipulam *strings* e buffers de forma insegura. Assim, identificou **5 Hits** e uma classificação de **nível de risco igual a 2**.

- Com a opção `-F`, o número de resultados é reduzido, removendo potenciais vulnerabilidades que foram classificadas como falsos positivos na primeira execução.
- Apesar da redução de alertas, ainda são destacadas funções como `strcpy()` e `sprintf()`, que continuam sendo reportadas como vulnerabilidades potenciais. Estas funções são reconhecidamente inseguras e frequentemente associadas a *buffer overflows*.

Como Resolver as Vulnerabilidades

Sugestões de correções para resolver as possíveis vulnerabilidades no código do ficheiro `mysig.c`:

1. Alterações para memcpy:

Substituir todas as ocorrências da função `memcpy` para as funções `memmove` ou `memcpy_s` (se disponível), ou assegurar que o tamanho do buffer de destino é suficiente antes de realizar a cópia.

Antes:

```
memcpy(destination, source, length);
```

Depois:

```
if (dest_size >= length) {  
    memmove(destination, source, length);  
}
```

2. Alterações para strcpy:

Substituir o uso das funções `strcpy` para `strncpy`, garantindo que a função não copie mais caracteres do que o tamanho permitido pelo *buffer* de destino.

Antes:

```
strcpy(destination, source);
```

Depois:

```
strncpy(destination, source, dest_size - 1);  
destination[dest_size - 1] = '\0'; // Assegura que a string termina em nulo
```

3. Validação do Tamanho das Alocações de Memória:

Para funções como `malloc`, deve-se adicionar verificações para assegurar que a alocação foi bem-sucedida antes de usar o *buffer* alocado.

```
char *buffer = (char *)malloc(size);  
  
if (buffer == NULL) {  
    perror("Erro de alocação de memória");  
    exit(EXIT_FAILURE);  
}
```

6. AFL

16. Use the AFL fuzzer tool to discover vulnerabilities in an extensive library made vulnerable by injecting a few bugs. While compiling the code, several applications are generated that call the library (see below). We will test the tiffcp application for this exercise.

O objetivo deste exercício é analisar com recurso de uma ferramenta de análise de

vulnerabilidades, designada por **AFL**, que irá permitir analisar de uma forma mais eficiente possíveis vulnerabilidades encontradas e porque essas vulnerabilidades existem. Neste caso, iremos fazer a análise de pelo menos três ficheiros de *input* que tenham causado *crash*.

Desta forma, procedeu-se ao recomendado no enunciado a execução de comandos nos referidos três passos. Na execução desses comandos houve o armazenamento dos executáveis no path `"/home/ss/apps/libtiff/work/bin"`. De seguida, foi necessário executar o **AFL** no SO na qual podemos observar na sua interface que existe a indicação de **"uniq crashes"**, isto que significa que a ferramenta criou entradas que causaram o *crash* da aplicação, que se encontraram em `"/home/ss/apps/libtiff/out_AFL/crashes/"`.

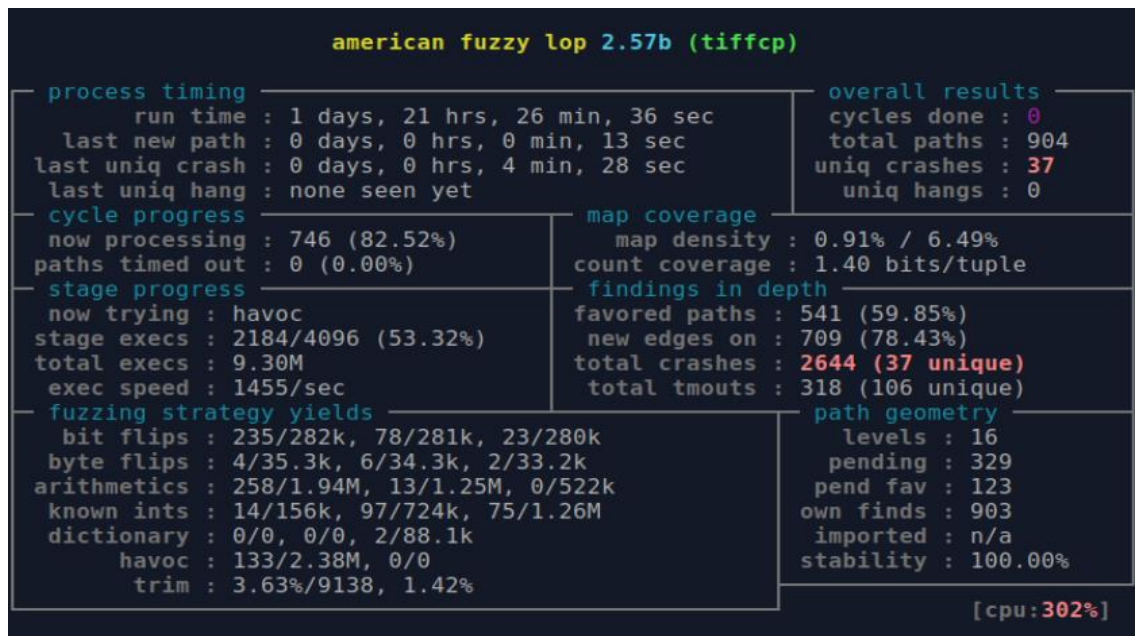


Figura 25

Tal como se pode observar na Figura 25, foi deixado em execução a ferramenta **AFL** em durante **1 dia e 21 horas**, onde se pode observar que esta identificou **37 uniq crashes**, ou seja,

vulnerabilidades, no total de **904 paths** explorados.

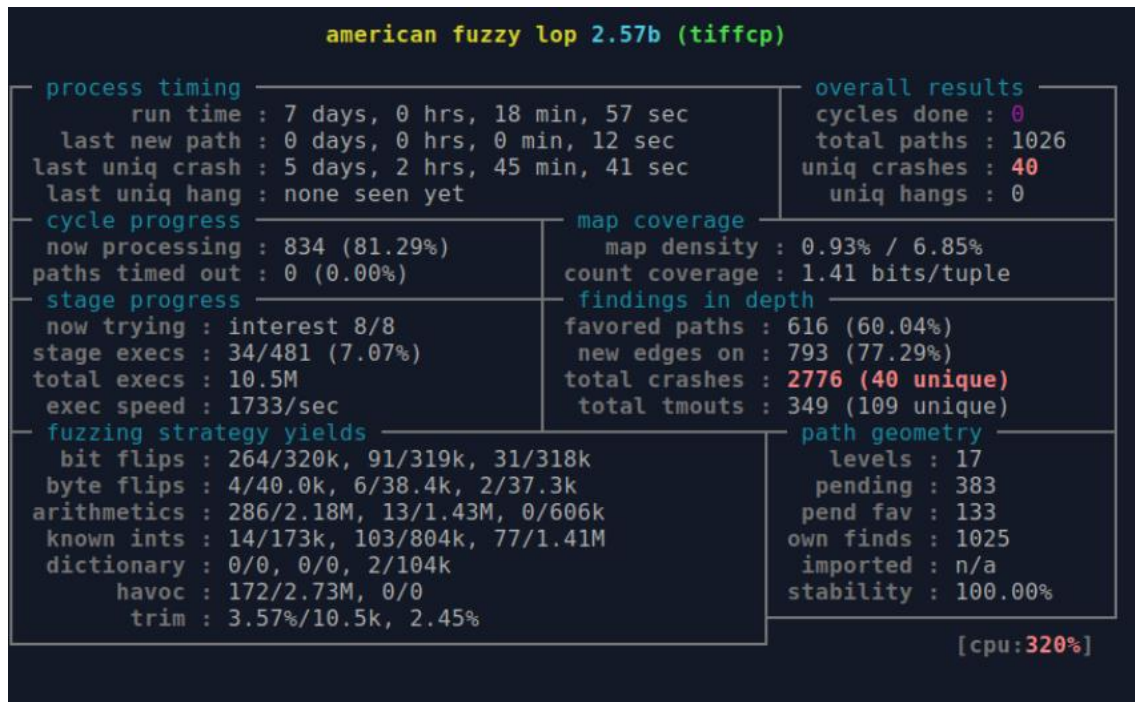


Figura 26

Para aumentar a probabilidade de a ferramenta encontrar ainda mais possíveis vulnerabilidades, para tal foi deixado em execução em durante **7 dia e 0 horas**, onde se pode observar que esta identificou **40 uniq crashes**, ou seja, vulnerabilidades, no total de **1026 paths** explorados, tal como se pode observar na Figura 26.

Para explorar cada uma das vulnerabilidades identificadas pela ferramenta decidimos utilizar o “gdb” como o mecanismo de *debug* e o do “tiffcp” como o executável para visualizar o que aconteceu a cada caso de teste e como a nossa *stack* que se foi alterando.

Dentro do mecanismo de *debug* usou-se o seguinte comando para cada caso de teste:

```
run -M /home/ss/apps/libtiff/out_AFL/crashes/crash_file_[i].tiff test[i].out
```

Vulnerabilidade 1:

```

ss@ss-VirtualBox:~/apps/libtiff/work/bin$ gdb ./tiffcp
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./tiffcp...done.
(gdb) run -M /home/ss/apps/libtiff/out_AFL/crashes/crash_file_1.tiff test.out
Starting program: /home/ss/apps/libtiff/work/bin/tiffcp -M /home/ss/apps/libtiff/out_AFL/crashes/crash_file_1.tiff test.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
TIFFReadDirectoryCheckOrder: Warning, Invalid TIFF directory; tags are not sorted in ascending order.
TIFFReadDirectory: Warning, Unknown field with tag 0 (0x0) encountered.
TIFFReadDirectory: Warning, Unknown field with tag 2449 (0x991) encountered.
TIFFReadDirectory: Warning, Unknown field with tag 52970 (0xceea) encountered.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff72402a7 in kill () at ../sysdeps/unix/syscall-template.S:78
78      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) where
#0  0x00007ffff72402a7 in kill () at ../sysdeps/unix/syscall-template.S:78
#1  0x000055555593bce in setExtraSamples (v=<synthetic pointer>, ap=<optimized out>, tif=0x555555914c20) at tif_dir.c:136
#2  TIFFVSetField (tif=0x555555914c20, tag=338, ap=<optimized out>) at tif_dir.c:396
#3  0x000055555599df1a in TIFFVSetField (ap=0x7ffff7fffd9e0, tag=338, tif=0x555555914c20) at tif_dir.c:905
#4  TIFFSetField (tif=tif@entry=0x555555914c20, tag=338) at tif_dir.c:849
#5  0x00005555555d51c5 in TIFFFetchNormalTag (tif=tif@entry=0x555555914c20, dp=dp@entry=0x555555915200, recover=recover@entry=0) at tif_dirread.c:5702
#6  0x00005555555f0b4d in TIFFReadDirectory (tif=tif@entry=0x555555914c20) at tif_dirread.c:3746
#7  0x000055555556345e8 in TIFFClientOpen (name=name@entry=0x7ffff7fffe259 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_1.tiff", mode=mode@entry=0x5555556e6866 "r",
clientdata=clientdata@entry=0x4, readproc=readproc@entry=0x555555654f10 <tiffReadProc>, writeproc=writeproc@entry=0x555555654c80 <tiffWriteProc>,
seekproc=seekproc@entry=0x555555654c40 <tiffSeekProc>, closeproc=0x555555654c00 <tiffCloseProc>, sizeproc=0x555555654800 <tiffSizeProc>,
mapproc=0x555555654980 <tiffMapProc>, unmapproc=0x5555556547b0 <tiffUnmapProc>) at tif_open.c:482
#8  0x0000555555565752 in TIFFFdOpen (mode=0x5555556e6866 "r", name=0x7ffff7fffe259 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_1.tiff", fd=4) at tif_unix.c:209
#9  TIFFOpen (name=0x7ffff7fffe259 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_1.tiff", mode=0x5555556e6866 "r") at tif_unix.c:248
#10 0x0000555555560ca3 in openSrcImage (imageSpec=0x7ffff7fffd50) at tiffcp.c:180
#11 main (argc=4, argv=0x7ffff7fffd5a8) at tiffcp.c:315
(gdb) ~

```

Figura 27

Na Figura 27 está representado o executável onde se passou como *input* o **crash_file_1.tiff**, que através da *stack* conseguimos ver que o programa aborta com **segmentation fault** quando é chamado o método **setExtraSamples()**. Nas Figuras 28 e 29 No debug conseguimos perceber que o *pointer* “v” apresenta a flag **syntectic pointer** em vez do valor para o qual ele aponta. Logo, existe um problema com este pointer (pois este possivelmente não aponta para nada ou aponta para vários lugares da *stack*) e assim consequentemente com o **td-> td_extrasamples** também, o que provoca um **segmentation fault**.

```

135 #ifdef MAGMA_ENABLE_CANARIES
136     MAGMA_LOG("AAH020", MAGMA_AND(td->td_transferfunction[0] != NULL, MAGMA_AND(td->td_samplesperpixel - *v > 1, \
137     !!(td->td_samplesperpixel - td->td_extrasamples > 1)))));
138 #endif
139
140     td->td_extrasamples = (uint16) *v;|
141     _TIFFSetShortArray(&td->td_sampleinfo, va, td->td_extrasamples);
142     return 1;

```

Figura 28

```

92 static int
93 setExtraSamples(TIFF* tif, va_list ap, uint32* v)

```

Figura 29

Vulnerabilidade 2:


```

(gdb) run -M /home/ss/apps/libtiff/out_AFL/crashes/crash_file_2.tiff test2.out
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ss/apps/libtiff/work/bin/tiffcp -M /home/ss/apps/libtiff/out_AFL/crashes/crash_file_2.tiff test2.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
TIFFReadDirectoryCheckOrder: Warning, Invalid TIFF directory; tags are not sorted in ascending order.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff72402a7 in kill () at ../sysdeps/unix/syscall-template.S:78
78      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) where
#0 0x00007ffff72402a7 in kill () at ../sysdeps/unix/syscall-template.S:78
#1 0x000055555559a082 in TIFFVSetField (tif=0x555555914c20, tag=277, ap=optimized out) at tif_dir.c:315
#2 0x000055555559d01a in TIFFVSetField (ap=0x7fffffdd9d0, tag=277, tif=0x555555914c20) at tif_dir.c:905
#3 TIFFSetField (tif=tif0entry=0x555555914c20, tag=277) at tif_dir.c:849
#4 0x000055555559d0e1 in TIFFFetchNormalTag (tif=tif0entry=0x555555914c20, dp=dp0entry=0x5555559151c0, recover=recover0entry=0) at tif_dirread.c:512
#5 0x0000555555515370 in TIFFReadDirectory (tif=tif0entry=0x555555914c20) at tif_dirread.c:3650
#6 0x0000555555563458 in TIFFClientOpen (name=name0entry=0x7fffffde258 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_2.tiff", mode=mode0entry=0x5555556e6866 "r", clientdata=clientdata0entry=0x4, readproc=readproc0entry=0x555555654f10 < TIFFReadProc>, writeproc=writeproc0entry=0x555555654c80 < TIFFWriteProc>, seekproc=seekproc0entry=0x555555654c40 < TIFFSeekProc>, closeproc=closeproc0x555555654c00 < TIFFCloseProc>, sizeproc=0x555555654800 < TIFFSizeProc>, mapproc=0x555555654980 < TIFFMapProc>, unmapproc=0x555555654780 < TIFFUnmapProc>) at tif_open.c:482
#7 0x0000555555565752 in TIFFFdOpen (mode=0x5555556e6866 "r", name=0x7fffffde258 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_2.tiff", fd=4) at tif_unix.c:209
#8 TIFFOpen (name=0x7fffffde258 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_2.tiff", mode=0x5555556e6866 "r") at tif_unix.c:248
#9 0x000055555559d0c3 in opensrcImage (imageSpec=0x7fffffdd640) at tiffcp.c:180
#10 main (argc=4, argv=0x7fffffde98) at tiffcp.c:315
(gdb)

```

Figura 30

Na Figura 30 está representado o executável onde se passou como *input* para o segundo caso de teste escolhido **crash_file_2.tiff**, e como é possível observar na Figura 30 ao analisar a nossa *stack* o programa dá **segmentation fault** devido a um problema encontrado na chamada da função **TIFFVSetField()**.

Vulnerabilidade 3:

```

(gdb) run -M /home/ss/apps/libtiff/out_AFL/crashes/crash_file_4.tiff test4.out
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ss/apps/libtiff/work/bin/tiffcp -M /home/ss/apps/libtiff/out_AFL/crashes/crash_file_4.tiff test4.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff72402a7 in kill () at ../sysdeps/unix/syscall-template.S:78
78      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) where
#0 0x00007ffff72402a7 in kill () at ../sysdeps/unix/syscall-template.S:78
#1 0x000055555559a082 in TIFFVSetField (tif=0x555555914c20, tag=277, ap=optimized out) at tif_dir.c:315
#2 0x000055555559d01a in TIFFVSetField (ap=0x7fffffdd9d0, tag=277, tif=0x555555914c20) at tif_dir.c:905
#3 TIFFSetField (tif=tif0entry=0x555555914c20, tag=277) at tif_dir.c:849
#4 0x000055555559d0e1 in TIFFFetchNormalTag (tif=tif0entry=0x555555914c20, dp=dp0entry=0x5555559151c0, recover=recover0entry=0) at tif_dirread.c:512
#5 0x0000555555515370 in TIFFReadDirectory (tif=tif0entry=0x555555914c20) at tif_dirread.c:3650
#6 0x0000555555563458 in TIFFClientOpen (name=name0entry=0x7fffffde258 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_4.tiff", mode=mode0entry=0x5555556e6866 "r", clientdata=clientdata0entry=0x4, readproc=readproc0entry=0x555555654f10 < TIFFReadProc>, writeproc=writeproc0entry=0x555555654c80 < TIFFWriteProc>, seekproc=seekproc0entry=0x555555654c40 < TIFFSeekProc>, closeproc=closeproc0x555555654c00 < TIFFCloseProc>, sizeproc=0x555555654800 < TIFFSizeProc>, mapproc=0x555555654980 < TIFFMapProc>, unmapproc=0x555555654780 < TIFFUnmapProc>) at tif_open.c:482
#7 0x0000555555565752 in TIFFFdOpen (mode=0x5555556e6866 "r", name=0x7fffffde258 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_4.tiff", fd=4) at tif_unix.c:209
#8 TIFFOpen (name=0x7fffffde258 "/home/ss/apps/libtiff/out_AFL/crashes/crash_file_4.tiff", mode=0x5555556e6866 "r") at tif_unix.c:248
#9 0x000055555559d0c3 in opensrcImage (imageSpec=0x7fffffdd640) at tiffcp.c:180
#10 main (argc=4, argv=0x7fffffde98) at tiffcp.c:315
(gdb)

```

Figura 31

Na Figura 31 está representado o executável onde se passou como *input* para o terceiro caso de teste foi escolhido **crash_file_4.tiff**, e como é possível observar na Figura 32 encontrámos uma situação semelhante à situação da vulnerabilidade 2. O programa dá **segmentation fault** devido a um problema encontrado na chamada da função **TIFFVSetField()**.

```

177 static int
178 TIFFVSetField(TIFF* tif, uint32 tag, va_list ap)

```

Figura 32

7. Large Language Models

17. Analyze manually the two codes and determine if they are vulnerable or secure to the above mentioned flaws. Your answer should justify your conclusions carefully.

Ao analisar manualmente os códigos dos ficheiros python **COLE.py** e **LEAKS.py**, podemos identificar as seguintes vulnerabilidades mencionadas:

1. **COLE.py – Vulnerabilidade de Path Traversal:**

- **Descrição da Vulnerabilidade:** A vulnerabilidade ocorre quando o código permite que o utilizador preparando a mensagem composta pelo nome da paste e o **gzip64encoded** para depois enviar pelos redis, possa introduzir um caminho manipulador que permita aceder a diretórios não autorizados. Se a variável `paste` (usada para construir caminhos de ficheiros) não for corretamente sanitizada, um atacante pode usar sequências como `../..` para aceder a ficheiros fora do diretório pretendido, acedendo a diretórios mais restritos.
- **Conclusão:** O código é vulnerável a ataques de **Path Traversal** porque não implementa medidas adequadas para sanitizar a entrada do utilizador. A falta de validação e remoção de componentes de caminho potencialmente perigosos permite o ataque.

2. **LEAKS.py – Vulnerabilidade de OS Command Injection:**

- **Descrição da Vulnerabilidade:** O código constrói comandos shell que incorporam entradas do utilizador (por exemplo, o **apk.package**) e executa-os com `os.system()`. Mesmo usando `quote()`, se a entrada não for corretamente validada, um atacante pode injetar comandos maliciosos.
- **Conclusão:** O código é vulnerável a **OS Command Injection** porque `os.system()` interpreta os comandos em contexto *shell*, e a entrada do utilizador não é suficientemente protegida contra a injeção de comandos. O uso de `subprocess.run()` seria uma alternativa mais segura.

18. Imagine that you are the developer of the two codes and would like ChatGPT to help you determine if they are vulnerable. It would be best if you designed the most appropriate question for ChatGPT so that it provides the most helpful response (this is called prompt engineering). In your report, you must explain the steps you have taken to create the most effective question for ChatGPT.

Primeiramente, foram-se fornecendo aos poucos pedaços do código para o **ChatGPT da OpenAI** perceber informações e o propósito da aplicação. Para o **COLE.py**, mostrou-se o protocolo do envio e processamento das mensagens as verificações que fazia e todas as variáveis existentes. Isto é importante pois cria uma linha de contexto para análises, deteções e sugestões que o AI pode fazer quanto ao programa.

Para obter uma resposta eficaz do **ChatGPT** sobre a identificação de vulnerabilidades no código, a formulação da questão é crucial. Aqui está como se faria a pergunta:

1. **Prompt Desenvolvido:**

Analise este código Python para verificar se contém vulnerabilidades de **Path Traversal** ou **OS Command Injection**. O código é o seguinte (adicionei as linhas específicas e o contexto relevante):

Código que inclui importações e linhas relevantes

Especificamente, gostaria de saber:

- Se existem riscos de segurança associados ao envio de mensagens e à variável `paste` em **COLE.py** e ao uso de `os.system()` em **LEAKS.py**.
- Se sim, indique as linhas específicas e explique como a vulnerabilidade pode ser explorada.
- Forneça uma solução de correção e explique o porquê de a solução ser eficaz.

2. **Explicação da Formulação:**

- **Por que é eficaz:** O *prompt* é direto e fornece contexto suficiente para que o **ChatGPT** possa responder de forma precisa. Especificar as linhas e o tipo de vulnerabilidade direciona a análise e garante uma resposta detalhada.
- **Considerações:** Incluir as importações e o código circundante ajuda a IA a entender melhor o ambiente do código. A pergunta também solicita uma explicação clara, o que é importante para uma análise compreensível.

Prompt versão inglesa para **COLE.py**:

“Is there a way the user can exploit this piece of code with path transversial? Show how the user can exploit this code, as well as the lines where the vulnerabilities reside and possible fixes to eliminate this concern and make the program safer and more secure.”

Estas *prompts* foram eficazes em originar uma boa resposta do **ChatGPT**, visto que indicou linhas de código onde estavam as vulnerabilidades, bem como a explicação de possíveis ataques usando essas linhas de código, e como poderíamos assegurar que este problema era controlado.

COLE.py

Vulnerability: Path Traversal Attack

Explanation:

In `COLE.py`, the script processes messages that include a `paste` variable, which is derived from external input without proper validation. This `paste` variable is used to construct file paths when saving files to disk. If an attacker can manipulate the value of `paste` to include directory traversal characters such as `../`, they can navigate out of the intended directory and write files to arbitrary locations on the file system.

Line Numbers and Code References:

- Line 46:

```
python
paste, gzip64encoded = splitted
```

- Line 58:

```
python
filename = os.path.join(PASTES_FOLDER, paste)
```

- Line 61:

```
python
if not os.path.exists(dirname):
    os.makedirs(dirname)
```

- Line 65:

```
python
with open(filename, "wb") as f:
    f.write(decoded)
```

How the Vulnerability Occurs:

- The `paste` variable is taken directly from the incoming `message` without any sanitization.
- It is then concatenated with `PASTES_FOLDER` to form `filename`, which determines where the file will be saved.
- If `paste` contains path traversal sequences like `../`, `os.path.join()` will interpret these and navigate to parent directories.
- This allows an attacker to write files outside of the intended `PASTES_FOLDER`.

Exploitation Scenario:

An attacker sends a crafted `message` where `paste` is set to `../../etc/passwd`, and `gzip64encoded` contains arbitrary data. The script decodes the data and writes it to the file path constructed from `paste`. As a result, the attacker can overwrite critical system files like `/etc/passwd` or place malicious files in sensitive directories.

Impact on the Program:

- Security Risks:**
 - Unauthorized creation or modification of files outside the intended directory.
 - Potential privilege escalation if `sys` ↓ files are overwritten.

LEAKS.py

Vulnerability: OS Command Injection

Explanation:

In `LEAKS.py`, the script constructs a shell command that includes user-controlled input without proper sanitization. Specifically, the `apk.package` variable, derived from the APK file's package name, is used in building the command that is executed using `os.system()`. If an attacker crafts an APK file with a malicious package name containing shell metacharacters or commands, they can inject arbitrary commands that the system will execute.

Line Numbers and Code References:

- Line 83:

```
python
dex = self.tempdir + "/" + self.apk.package + ".dex"
```

- Lines 88-89:

```
python
comm = "%s" % (" ".join(quote(arg) for arg in args))
os.system(comm)
```

How the Vulnerability Occurs:

- The `apk.package` value is derived from the APK file and can contain arbitrary strings.
- This value is included in the `dex` file path and then in the command arguments.
- Although `quote()` is used, if not handled correctly, shell metacharacters in `apk.package` can break out of the quoted context.
- Using `os.system()` executes the command in the shell, which interprets special characters, leading to command injection.

Exploitation Scenario:

An attacker provides an APK file with a package name like `com.example; rm -rf /`. When the script constructs the command, it becomes:

```
shell
jadx com.example; rm -rf /.dex -d /tmp/leaks-xxxx --deobf
```

The shell interprets the semicolon as a command separator, executing `rm -rf /`, which deletes all files on the system the script has permissions to.

Impact on the Program:

- Security Risks:**
 - Execution of arbitrary commands, leading to system compromise.
 - Deletion or corruption of data.
 - Unauthorized access to sensitive information.
- Functional Risks:**
 - The script may crash or behave unexpectedly due to manipulated inputs.
 - Potential denial of service if `sys` ↓ sources are affected.

Figura 33

Figura 34

19. Ask ChatGPT to provide you with a fix/patch to the codes that were considered vulnerable. Again, you must explain in detail the steps you have taken to obtain the best response from ChatGPT. You must also discuss the reactions you have received, namely, whether they were appropriate.

Para obter a correção das vulnerabilidades, faria a seguinte pergunta ao ChatGPT:

1. Prompt para Correção:

"Forneça uma solução para corrigir as seguintes vulnerabilidades no código Python:

COLE.py: Vulnerabilidade de Path Traversal na variável `paste`. O código atual é:

```
paste, gzip64encoded = splitted  
# Código original continua...
```

LEAKS.py: Vulnerabilidade de OS *Command Injection* ao usar `os.system()`. O código atual é:

```
comm = "%s" % (" ".join(quote(arg) for arg in args))  
os.system(comm)
```

- Explique detalhadamente como as correções funcionam e por que são eficazes. Sugira melhores práticas para evitar vulnerabilidades semelhantes no futuro."

2. Discussão das Respostas:

- **Reação e Avaliação:** Avaliar se o ChatGPT sugere correções que aderem às melhores práticas de segurança, como a validação de entradas e o uso seguro de módulos como `subprocess`.
- **Ajustes ao Prompt:** Se a resposta for genérica ou incorreta, ajustaria o *prompt* para ser mais específico, como fornecer mais detalhes sobre o que já sabemos sobre a vulnerabilidade (por exemplo, impacto potencial e exemplos de exploração).

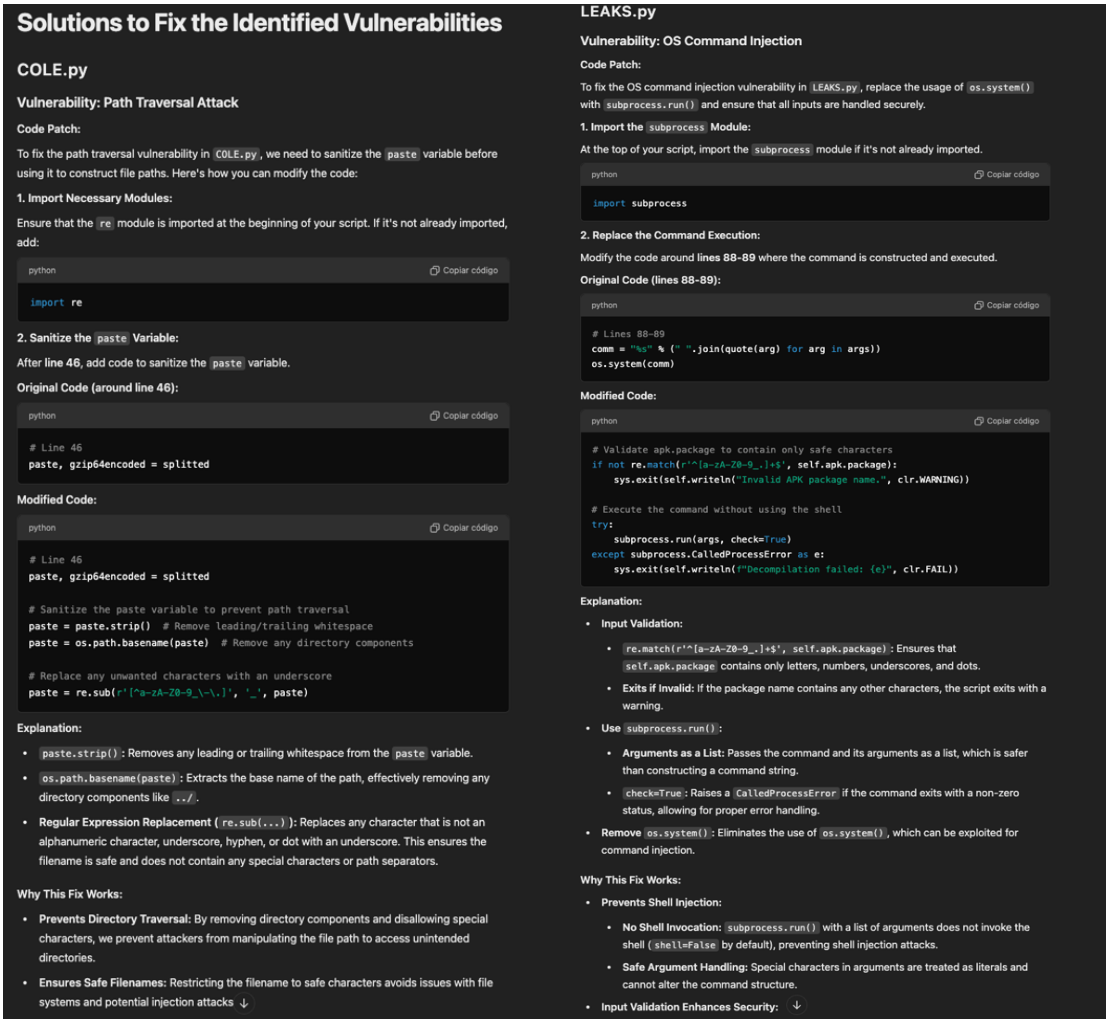


Figura 35

Figura 36