



**Ciências**  
**ULisboa**

| Informática

# **Sistemas de Software Seguros (SSS)**

Mestrado em Segurança Informática

## **Segurança de Software (SS)**

Mestrado em Engenharia Informática

Mestrado em Informática

**2024/2025**

# **Projeto**

Gustavo Henriques Nº 64361

Leonardo Monteiro Nº 58250

Maria Figueirinhas Nº 46494

# Índice

Primeira Parte.....	4
2. vulnApp .....	4
01 – Characterize the attack surface of each application.....	4
01.1 – Map_path .....	4
01.2 – MIME .....	5
01.3 – Mysig.....	5
02 – Map_path ( buffer overflows ).....	5
03 – Map_path ( input that exploits the identified vulnerabilities ) .....	6
04 – Mime ( buffer overflows ).....	6
05 – Mime ( solution ) .....	7
06 – Mysig ( integer overflow ).....	7
07 – Mysig ( solution ) .....	7
3. SSS-DB .....	8
08 – SQL Injection.....	8
09 – Stored XSS Attack .....	8
10 – Reflected XSS Attack .....	9
11 – Reflected XSS Attack ( solution ) .....	10
12 – Command injection.....	11
13 – Command injection ( solution ) .....	11
Segunda Parte.....	12
4. FlawFinder .....	12
14 - Error report.....	12
15 - Same command with the -F option .....	13
6. AFL .....	14
16 - Using the AFL fuzzer tool to discover vulnerabilities .....	14
7. Large Language Models.....	14
17 – Analysing if each code is vulnerable or secure.....	14
17.1 – COLE.py.....	14
17.2 – LEAKS.py .....	14
18 – Prompt engineering with ChatGPT .....	15
18.1 – COLE.py.....	15
18.2 – LEAKS.py .....	16
19 – Ask ChatGPT to fix/patch the codes.....	16

19.1 – COLE.py.....	16
19.2 – LEAKS.py .....	18

## ÍNDICE DE FIGURAS

FIGURA 1. CODE EXECUTION RESULT FOR MAP_PATH ATTACK.	6
FIGURA 2. SOLUTION FOR THE SQL INJECTION.	8
FIGURA 3. RESULTS OF THE SQL INJECTION ATTACK.	8
FIGURA 4. RESULTS OF THE STORED XSS ATTACK.	9
FIGURA 5. SOLUTION OF THE REFLECTED XSS ATTACK.	9
FIGURA 6. RESULT OF THE REFLECTED XSS ATTACK.	10
FIGURA 7. FIELD OF INPUT THAT CAN SUFFER A COMMAND.	11
FIGURA 8. FILE TO BE ATTACKED.	11
FIGURA 9. INPUT SOLUTION FOR THE COMMAND INJECTION.	11
FIGURA 10. RESULTS OF THE FLAWFINDER REPORT FOR MYSIG.	12

# Primeira Parte

## 2. vulnApp

### 01 – Characterize the attack surface of each application.

#### 01.1 – Map\_path

A *attack surface* do *map\_path* é o conteúdo do ficheiro *input* (file system). Uma vez que nada dentro do programa verifica se o utilizador tem acesso às diretorias indicadas pelo ficheiro *input* no terminal, correr o programa permite a entrada em qualquer diretoria desejada. A partir daí, é possível entrar onde o atacante quiser e, eventualmente, aceder aos ficheiros aí presentes. Aqui, a função vai concatenando o caminho dos diretórios à variável *mapped\_path*, sem fazer uma verificação de segurança adequada no conteúdo de *dir*. Isto permite que um atacante insira partes do caminho como *".."* para manipular o caminho e aceder a diretórios superiores.

```
strcat(mapped_path, "/");
strcat(mapped_path, dir);

if ((ret = chdir(mapped_path)) < 0){
    printf("couldn't chdir to %s !\n", mapped_path);
    strcpy(mapped_path, old_mapped_path);
}
```

Aqui, a função tenta mudar o diretório para o valor acumulado em *mapped\_path*, o que pode ser controlado por um atacante se *orig\_path* tiver valores manipulados. Isto abre a possibilidade de mudar entre diretórios, permitindo acesso a áreas não autorizadas do sistema de ficheiros.

```
f = fopen(argv[1], "r");
fgets(orig_path, MAXPATHLEN + 20, f);
fclose(f);

map_dir_chdir(orig_path);
```

O código lê o caminho original de um ficheiro (através de *fopen* e *fgets*) e depois tenta mudar de diretoria para esse caminho com *map\_dir\_chdir*. Se um atacante puder modificar o conteúdo do ficheiro entre a abertura e a leitura,

ou se manipular o sistema de ficheiros entre as chamadas a *fopen* e *chdir*, este pode alterar o comportamento do programa, levando a um aumento de privilégios ou à execução de operações indesejadas.

## 01.2 – MIME

A superfície de ataque do MIME é o conteúdo do ficheiro de entrada. Este é um tipo de ataque de files system /User interface. O programa lê diretamente o arquivo fornecido pelo utilizador, sem validar adequadamente o conteúdo, permitindo que um atacante forneça um ficheiro malicioso. Isso pode resultar em ataques como **Path Traversal**, **execução de código malicioso** ou **negação de serviço (DoS)** através de ficheiros manipulados. A linha de código que permite isso é:

```
temp = fopen(argv[1], "r");
```

## 01.3 – Mysig

Agora em relação ao *mysig*. Este código implementa uma parte de um servidor DNS (Domain Name System), responsável por lidar com consultas de resolução de nomes de domínio. O código constrói pacotes de consulta DNS, empacotando informações como nomes de domínio em um formato adequado para serem enviados a servidores DNS. Além disso, o servidor envia essas consultas para servidores específicos e processa as respostas que recebe, extraindo registos de recursos, como endereços IP, associados aos domínios consultados. O código faz uso de alocação de memória dinâmica para armazenar temporariamente dados, como os nomes de domínio e as respostas DNS, e realiza operações de manipulação de strings e buffers. Em resumo, a principal função do código é atuar como um cliente de resolução DNS, montando e enviando consultas, recebendo as respostas e retornando informações sobre os domínios consultados.

No caso deste código, a attack surface está principalmente relacionada ao sistema operacional, especialmente devido à manipulação de memória dinâmica, interações com o sistema de arquivos e rede e uso de funções para manipulação de strings e buffers. O código contém interações com o sistema de arquivos (por exemplo, com *malloc* para alocar buffers para armazenar dados de rede) e com a rede (enviando pacotes DNS). Em sistemas operacionais, essa interação pode ser uma linha de ataque, pois o acesso a essas camadas envolve manipulação de buffers e de dados externos que podem ser comprometidos. Para além, como o código usa funções como *strcpy*, *memcpy* e *dn\_expand* podem existir vulnerabilidades de buffer overflow devido a estas funções não fazerem verificações de limite de tamanho.

## 02 – Map\_path ( buffer overflows )

No programa *map\_path*, há cinco *buffer overflows*. Acontecem nas linhas 30, 68, 69, 83, 86 e 112. Na linha 30 e 86, ocorrem pois está a ser utilizada a função *strcpy* (copia uma string). O problema de utilizar esta função é que não é feita nenhuma verificação de tamanho da string que vai ser copiada em relação ao local de destino.

Na linha 30, *strcpy* copia *mapped\_path* para *path*. A variável *mapped\_path* é uma string contida dentro do input de main, que representa o caminho passado como argumento. Esta variável pode dar overflow no buffer criado para *path*.

```
30 strcpy(path, mapped_path);
```

Caso *mapped\_path* seja maior ou igual a *path* (\0 será acrescentado no final da string, causando buffer overflow nestes casos também), ocorrerá overflow.

A função *strcat* concatena strings sem verificar se o tamanho do resultado ultrapassa o espaço reservado para a operação. É este o caso na linha 68 e 69, com as seguintes linhas:

```
68.strcat ( mapped_path, "/" );
```

```
69.strcat ( mapped_path, dir );
```

A variável *mapped\_path* é concatenada com “/” e depois com *dir*. O resultado pode eventualmente originar um *buffer overflow*, dependendo do input inicial.

Na linha 83, o *buffer overflow* ocorre devido a um *strcpy*, de acordo com o código seguinte:

```
83.strcpy (old_mapped_path, mapped_path);
```

A variável *old\_mapped\_path* está definida na linha 36 como sendo um *buffer* de tamanho igual a *MAXPATHLEN*. A variável *mapped\_path* é um *buffer* com tamanho igual a *MAXPATHLEN* também, inicializada apenas contendo “/”. No entanto, a partir da linha 41, há uma série de verificações sobre o conteúdo de *dir* e de *mapped\_path*, culminando no possível *buffer overflow* mencionado nas linhas 68 e 69. A variável *mapped\_path* pode já estar em *buffer overflow* por esta fase, dado que *dir* é fornecido pelo utilizador e o seu tamanho nunca é verificado. Ao copiar esta string em possível *overflow* para o *old\_mapped\_path*, cujo tamanho é fixo, e, usando *strcpy*, que não verifica o tamanho da string a ser copiada, então há a forte possibilidade de gerar um *buffer overflow* também nesta variável.

Na linha 86, a variável *orig\_path* é copiada para a variável *path*. Esta variável é definida na linha 84 como sendo um *pointer* para a lista *pathspace*, que tem de tamanho *MAXPATHLEN*, definido, na linha 35. *orig\_path* é definida na linha 168, na *main*, com tamanho superior a *MAXPATHLEN* e o seu conteúdo é o caminho original para o ficheiro em questão, que pode ser maior que *MAXPATHLEN*. Uma vez que a cópia é feita utilizando *strcpy*, não é feita nenhuma verificação de tamanho em relação ao que é copiado e para onde, pelo que pode acontecer um *buffer overflow* como resultado.

### 03 – Map\_path ( input that exploits the identified vulnerabilities )

Para atacarmos as vulnerabilidades do *Map\_path* criamos um ficheiro teste com mais de 20 caracteres que era o máximo de caracteres que as variáveis podiam ter ( 19 caracteres mais o /0 ). E com isto causamos um *buffer overflow* tendo o código mudado para a diretoria que metemos no input e corrompido a pilha de execução do programa, como podemos ver na imagem abaixo:

Figura 1. Code execution result for map\_path attack.

```
gustavo@gustavo:/mnt/c/Users/gusta/OneDrive/Ambiente de Trabalho/Mestrado - FCUL/SS/Projeto/map_path$ ./map_path teste
Current directory = /mnt/c/Users/gusta/OneDrive
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

### 04 – Mime ( buffer overflows )

Este programa é um decodificador básico para mensagens MIME codificadas em base64. Lê um arquivo especificado pelo utilizador, verifica se está codificado em base64 (usando o cabeçalho MIME), e, se estiver, decodifica o conteúdo de volta para um formato de 8 bits.

Possível *buffer overflow*:

```
136. if (*fbuff++ == '\n' || fbuf >= &fbuf[MAXLINE]) {
...
}
```

Na função *MIME\_func*, o ponteiro *fbuff* é utilizado para gravar dados decodificados no *buffer fbuf*. Este *buffer* tem um tamanho fixo, definido por *MAXLINE + 1*. *fbuff* começa por apontar para o início de *fbuf*, mas é continuamente incrementado à medida que os dados são gravados. O problema está na verificação indireta e inadequada que o código faz para garantir que *fbuff* não ultrapasse o tamanho do *buffer*. A condição atual verifica apenas o ponteiro base *fbuf*, e não *fbuff*, o que permite que *fbuff* continue a ser incrementado e acabe por escrever fora dos limites de *fbuf*. Isso causa uma vulnerabilidade de *buffer overflow*, permitindo que o programa escreva em áreas não alocadas da *heap*, corrompendo dados e afetando variáveis e estruturas adjacentes. Esta vulnerabilidade pode ser explorada por um invasor para alterar o comportamento do programa, escrevendo fora do espaço alocado para o programa.

## 05 – Mime ( solution )

Para evitar o *buffer overflow*, precisamos garantir que o ponteiro *fbufp* nunca ultrapasse o limite do *buffer fbuf*. A sugestão é alterar a condição de verificação para comparar o *fbufp* diretamente com o final do *buffer*.

Em vez disto:

```
if (*fbufp++ == '\n' || fbuf >= &fbuf[MAXLINE]) {
```

Fazer:

```
if *(fbufp - 1) == '\n' || fbufp >= &fbuf[MAXLINE]) {
```

Esta versão garante que *fbufp* não ultrapassa o limite de *fbuf*: (*fbufp* >= &*fbuf*[*MAXLINE*]) impede que *fbufp* aponte para fora do *buffer*. Do mesmo modo, a primeira condição na solução verifica se o último carácter dentro de *fbuf* antes da posição atual é o newline (\n). Isto serve para ter a certeza de que o *buffer* está a ser resetted na altura correta.

## 06 – Mysig ( integer overflow )

Na função *RRextract*, o valor de *dlen* (comprimento dos dados do RR) é extraído diretamente da mensagem DNS recebida: *GETSHORT(dlen, cp)*; aqui, *dlen* é extraído de uma posição na mensagem DNS (o ponteiro *cp*), que está fora do controlo da aplicação. Isso significa que um atacante pode manipular esse valor no pacote DNS para ser maior do que o esperado, causando assim um *integer overflow*. Após ler *dlen*, o código tenta garantir que a área referida por *cp* tem espaço suficiente para ler os dados, usando a macro *BOUNDS\_CHECK*: *BOUNDS\_CHECK(cp, dlen)*; A macro *BOUNDS\_CHECK* verifica se o ponteiro *cp* mais o valor *dlen* não ultrapassa o fim da mensagem (*com*), mas isso não verifica se o *buffer* onde os dados serão armazenados tem espaço suficiente para os conter.

Esta verificação é limitada à estrutura da mensagem DNS, mas não protege contra um *buffer overflow* quando os dados são copiados ou processados. A cópia dos dados do RR para o *buffer* é feita logo após a verificação de limites, sem verificar se o espaço disponível no *buffer* é suficiente para os dados que estão a ser copiados: *rdatap = cp*;

Neste ponto, *rdatap* (um ponteiro para a posição atual no *buffer*) é configurado para apontar para os dados que serão lidos de *cp*. Se o valor de *dlen* for maior do que o espaço disponível no *buffer*, a função poderá copiar mais dados do que o *buffer* suporta, causando um *buffer overflow* nesta variável *rdatap*.

## 07 – Mysig ( solution )

Uma alteração que pode impedir a utilização desta vulnerabilidade é verificar o valor de *dlen* antes de o usar:

```
if (dlen > sizeof(data)) {  
    printf("Data length exceeds buffer size\n");  
    hp->rcode = FORMERR; // Código de erro  
    return -1;  
}
```

Isso garantiria que qualquer operação de leitura ou escrita não excederia o tamanho do *buffer*, prevenindo o *buffer overflow*.

### 3. SSS-DB

#### 08 – SQL Injection

No nível de segurança 0, o código permite a execução de *SQL Injection* porque as variáveis não são escapadas ou saneadas antes de serem usadas na *query*. Este é um caso típico de vulnerabilidade onde um utilizador mal-intencionado pode explorar essa fraqueza para obter acesso não autorizado ao sistema ou até mesmo manipular os dados na base de dados.

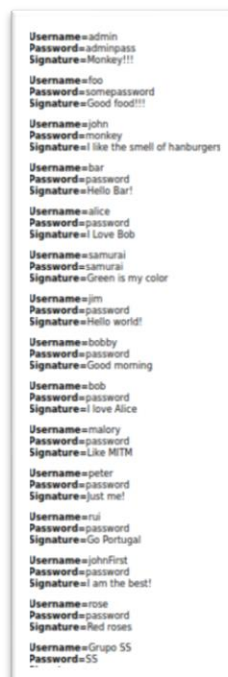
```
$lQuery = "SELECT * FROM accounts WHERE username='".  
$lUsername .  
"' AND password='".  
$lPassword .  
"'";
```

Ao analisar este código SQL é possível fazer uma SQL Injection ao meter no Username a seguinte string:

' OR '1'='1' --

Figura 2. Solution for the SQL Injection.

Ao inserir isto conseguimos obter a seguinte lista de nomes:



```
Username=admin  
Password=adminpass  
Signature=Monkey!!!  
  
Username=foo  
Password=somepassword  
Signature=Good food!!!  
  
Username=john  
Password=monkey  
Signature=I like the smell of hamburgers  
  
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=I 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  
  
Username=Grupo SS  
Password=SS
```

Figura 3. Results of the SQL Injection attack.

#### 09 – Stored XSS Attack

Durante a criação de um novo utilizador, não ocorre qualquer saneamento do *input* que é fornecido pelo utilizador antes de exibir a página onde este fica guardado. Isto torna a opção *Show Log* vulnerável a ataques de *stored XSS*. Um atacante pode colocar no username um script malicioso que será guardado e chamado de novo quando qualquer utilizador aceder à página do *Show Log* e então, será executado como parte do script da página.

Para realizar o ataque, no momento de criação de user, no campo de input Username, inserimos o seguinte input:

```
<img src=x onerror=alert(&#39;hello&#39;)>
```



Com isto criamos um *XSS stored attack* que irá fazer com que apareça um *popup*, tanto no momento de criação de *user* como quando se tenta procurar o nome do utilizador na opção *Show Log* no *Core Control*.

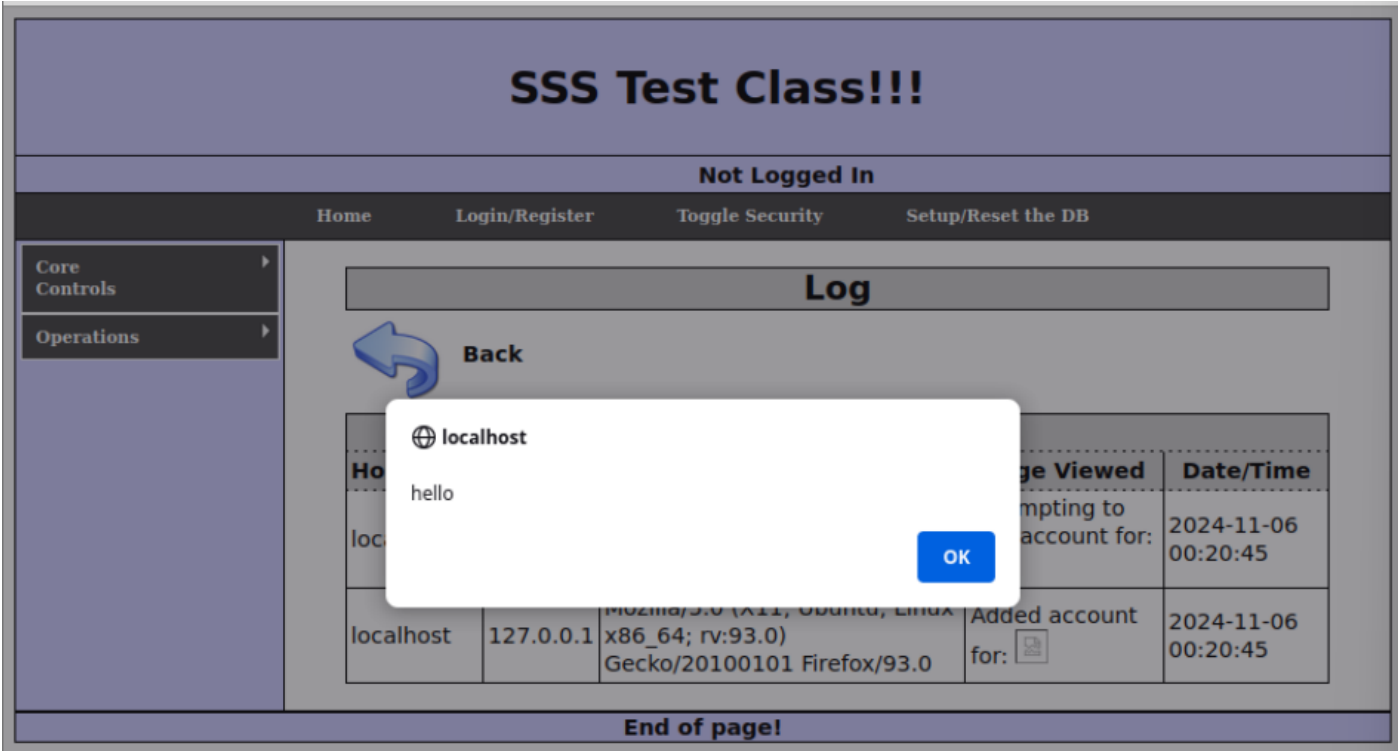
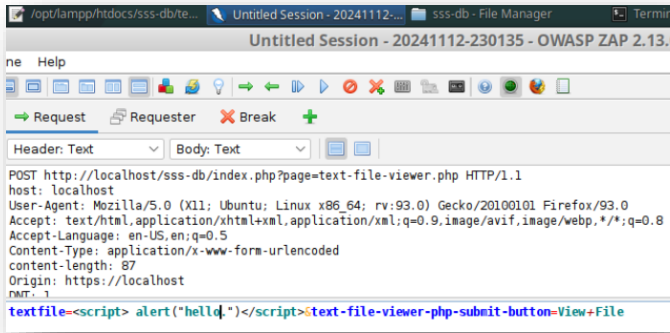


Figura 4. Results of the stored XSS attack.

## 10 – Reflected XSS Attack

O ataque funciona alterando o nome do textfile no request que o cliente faz ao servidor. O servidor não faz qualquer saneamento do request e simplesmente corre o nome do .txt como um script, criando o pop up.

Figura 5. Solution of the Reflected XSS attack.



## 11 – Reflected XSS Attack ( solution )

A vulnerabilidade existe porque o parâmetro "textfile" não é corretamente validado ou escapado antes de ser utilizado na página web. Quando o utilizador seleciona um ficheiro, o valor do parâmetro "textfile" é incluído diretamente na resposta HTML sem qualquer sanitização. Isto permite que um atacante manipule a requisição, inserindo um script malicioso que será executado no navegador do utilizador quando a página for carregada. Para mitigar esta vulnerabilidade, a aplicação deve garantir que qualquer entrada do utilizador, como o valor do parâmetro "textfile", seja rigorosamente validada e escapada antes de ser renderizada no HTML.

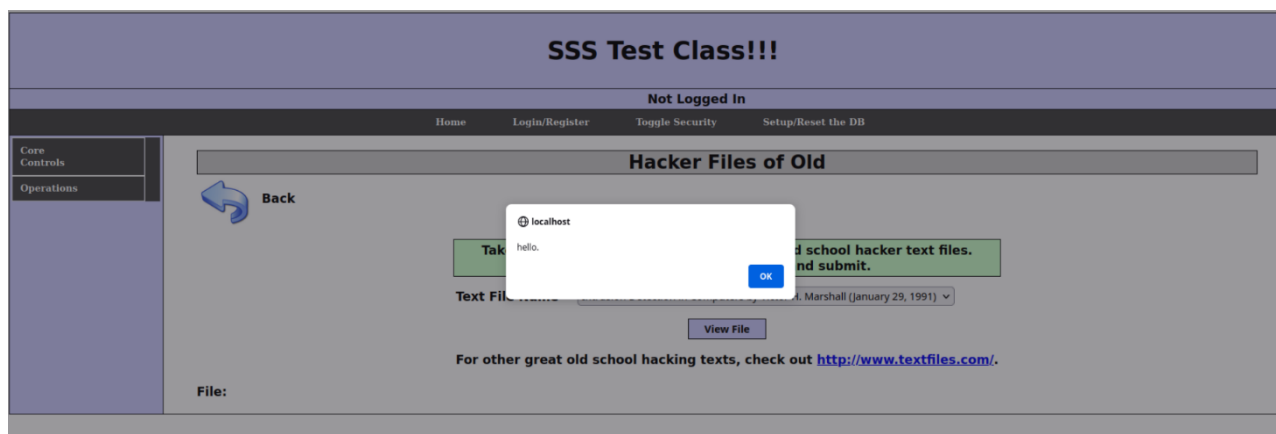


Figura 6. Result of the Reflected XSS attack.

Em vez de permitir que o parâmetro "textfile" contenha qualquer valor, a aplicação poderia restringi-lo a identificadores específicos que correspondam a ficheiros válidos. Assim, ao invés de aceitar diretamente caminhos ou nomes de ficheiros, o sistema deveria utilizar um esquema de IDs para representar os ficheiros, reduzindo a possibilidade de injeção de código malicioso. Além disso, a implementação de tokenização para níveis de segurança mais altos contribui para proteger o sistema, uma vez que assegura que o valor do parâmetro apenas corresponda a números predefinidos que representam ficheiros legítimos, prevenindo um reflected XSS attack.

## 12 – Command injection

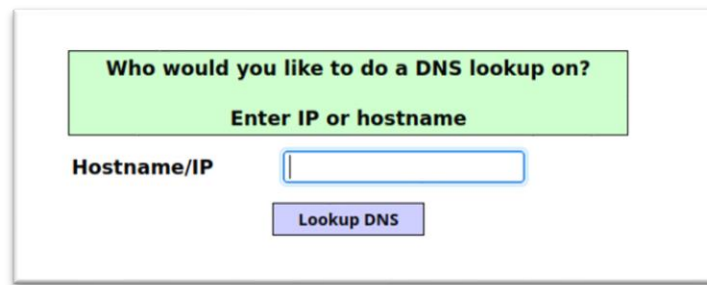
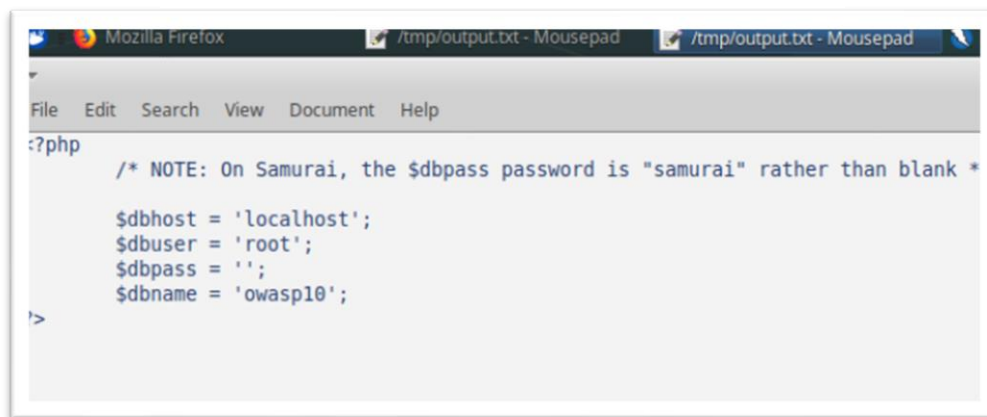


Figura 7. Field of input that can suffer a command.



```
/* NOTE: On Samurai, the $dbpass password is "samurai" rather than blank */  
$dbhost = 'localhost';  
$dbuser = 'root';  
$dbpass = '';  
$dbname = 'owasp10';  
>
```

Figura 8. File to be attacked.

Para fazer este ataque descobrimos que o ficheiro que queremos é o config.inc, mostrado em cima. Com isto, primeiramente tentamos diretamente imprimir este ficheiro com o comando “; cat /opt/lampp/htdocs/sss-db/config.inc” mas, como o código do ficheiro está em php não era possível ele ser impresso na aplicação. Para solucionar este problema, decidimos copiar o código que estava neste ficheiro mas sem a primeira e última linha, de seguida, bastava imprimir este ficheiro e desta maneira conseguimos com que aparecesse na app, como podemos ver na imagem abaixo. Assim, conseguimos arranjar uma maneira de ver o conteúdo deste ficheiro sem sequer utilizar o



```
Results for ; tail -n +2 /opt/lampp/htdocs/sss-db/config.inc | head -n -1 > /tmp/attackFile.txt; cat /tmp/attackFile.txt;  
  
/* NOTE: On Samurai, the $dbpass password is "samurai" rather than blank */  
$dbhost = 'localhost';  
$dbuser = 'root';  
$dbpass = '';  
$dbname = 'owasp10';
```

Figura 9. Input solution for the command injection.

proxy:

## 13 – Command injection ( solution )

A vulnerabilidade de injeção de comandos existe porque a aplicação passa o valor do parâmetro "target\_host" diretamente para o comando dnslookup sem a devida sanitização, especialmente nos níveis de segurança baixos. Nos níveis de segurança 0 e 1, o valor de "target\_host" é obtido diretamente através de \$\_REQUEST e não é validado, o que permite a injeção de caracteres especiais como ; ou & para encadear comandos adicionais. Assim, um atacante pode manipular o campo "target\_host" para executar comandos no servidor, obtendo, por exemplo, informações sensíveis como as credenciais de acesso à base de dados, explorando a vulnerabilidade para executar comandos arbitrários.

Para proteger a aplicação contra este tipo de ataque, é crucial validar e sanitizar adequadamente os inputs do utilizador. Uma forma eficaz é permitir apenas valores específicos, como endereços IP ou nomes de domínio válidos,

utilizando expressões regulares para verificar o formato do input, o que já é feito nos níveis de segurança mais altos. Além disso, a aplicação deve evitar o uso direto de funções que executam comandos do sistema operacional com inputs fornecidos pelo utilizador. Em vez disso, poderia recorrer a APIs ou bibliotecas específicas para resolver DNS sem necessidade de comandos externos, mitigando o risco de injeção de comandos.

## Segunda Parte

### 4. FlawFinder

#### 14 - Error report

Figura 10. Results of the FlawFinder report for Mysig.

```
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:456: [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:506: [2] (buffer) char:
Statically-sized arrays can be improperly restricted, leading to potential
overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use
functions that limit length, or ensure that the size is larger than the
maximum possible length.
../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, strcpy_s, or strncpy (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, strcpy_s, or strncpy (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, strcpy_s, or strncpy (warning: strncpy
easily misused). Risk is low because the source is a constant string.
```

Error	Rating	Why
Mysig 402  memcpy(cp1, cp, 18);	FALSE POSITIVE	O tamanho que estamos a passar buffer de destino (cp1) é validado através do BOUNDS_CHECK na linha anterior. Logo apesar de nesta linha não ser feita nenhuma verificação, pela linha anterior conclui-se que esta vulnerabilidade não pode ser explorada.
Mysig 456  memcpy(cp1, cp, n);	TRUE POSITIVE	O tamanho que estamos a passar para o cp1 pode ser maior do que esta variável consegue aceitar, causando assim um bufferoverflow.
Mysig 506  char exp_dn[200], exp_dn2[200];	FALSE POSITIVE	As verificações de tamanho antes de acessar o buffer, garantem que não há risco de overflow.

Mysig 524 <pre>strcpy(temp, "HEADER JUNK:");</pre>	FALSE POSITIVE	O tamanho do buffer é maior do que a string copiada. Temp é um char pointer com capacidade para 400 caracteres e a string que estamos a copiar tem 12 caracteres.
Mysig 531 <pre>strcpy(exp_dn, "lcs.mit.edu");</pre>	FALSE POSITIVE	O tamanho do buffer é maior do que a string copiada. Exp_dn é um char com capacidade para 200 caracteres e a string que estamos a copiar tem 11 caracteres.
Mysig 587 <pre>strcpy(exp_dn2, "ccs.ocs.fcul.pt");</pre>	FALSE POSITIVE	O tamanho do buffer é maior do que a string copiada. Exp_dn2 é um char com capacidade para 200 caracteres e a string que estamos a copiar tem 14 caracteres.

## 15 - Same command with the -F option

### 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:456: [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, strcpy_s, or strncpy (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, strcpy_s, or strncpy (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, strcpy_s, or strncpy (warning: strncpy
easily misused). Risk is low because the source is a constant string.
```

A ferramenta Flawfinder, ao executar o comando com a opção -F, filtra potenciais False Positives (FP), mas nem sempre de forma precisa. Observámos que, a linha 506 identificada anteriormente como False Positives foi corretamente removida do output. Contudo, as outras que também analisámos como FP (402, 524, 531, 587) permanecem no resultado, mostrando que a ferramenta não consegue identificar consistentemente todas as situações que não representam vulnerabilidades reais. Com isto, concluímos que a ferramenta não é 100% precisa na distinção entre True Positives e False Positives. Este comportamento é esperado, pois Flawfinder é uma análise estática que não compreende totalmente o contexto do código.



## 6. AFL

### 16 - Using the AFL fuzzer tool to discover vulnerabilities

## 7. Large Language Models

### 17 – Analysing if each code is vulnerable or secure

#### 17.1 – COLE.py

O código COLE.py parece estar vulnerável a uma falha de path traversal.

No código, a variável `paste` parece conter um caminho de arquivo. Esse caminho é processado na seguinte linha:

```
file_name_paste = paste.split('/')[ -1]
```

Aqui, o nome do arquivo é extraído a partir do caminho fornecido em `paste`. No entanto, o código não faz nenhuma validação ou sanitização para garantir que o caminho não contenha sequências como `../`, que poderiam permitir que um atacante navegasse para diretórios fora do diretório autorizado.

Se o utilizador fornecer um caminho como `../etc/passwd`, o código processará essa entrada sem qualquer verificação, o que poderia resultar em acesso a arquivos fora do diretório de trabalho desejado. Isso caracteriza uma vulnerabilidade de path traversal, onde o atacante pode acessar arquivos sensíveis ou até mesmo sobrescrever arquivos importantes.

#### 17.2 – LEAKS.py

O código LEAKS.py parece estar vulnerável a uma falha de Command Injection e pode ser identificada na seguinte linha de código:

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

Aqui, o comando **comm** é construído a partir da lista de argumentos **args**, que inclui o caminho para o binário **jadx**.

```
args = [self.jadx, dex, "-d", self.tempdir, "--deobf"]
```

Esse comando é executado com **os.system(comm)**. A vulnerabilidade ocorre porque o valor de **self.jadx(/path/to/jadx)** e **dex** (que é o nome do pacote APK obtido por **self.apk.package(package\_name)**) pode ser manipulado. Se um atacante conseguir modificar o **dex** ou o caminho do binário **jadx**, ele pode injetar caracteres especiais, como **;**, **&**, **|** ou **../**, permitindo a execução de comandos maliciosos.

```
/path/to/jadx package_name; rm -rf /.dex -d /tempdir --deobf
```

O comando **/path/to/jadx package\_name** tenta rodar o binário **jadx (/path/to/jadx)** no arquivo **package\_name**, mas um atacante pode manipular o nome do pacote para incluir **; rm -rf /** (ficando assim **/path/to/jadx package\_name ; rm -rf /**), injetando um comando malicioso. O **;** permite adicionar outro comando, e o **rm -rf /** excluiria todos os arquivos do sistema. Os parâmetros **.dex -d /tempdir --deobf** são ignorados, pois o **rm** é executado primeiro.

Esta vulnerabilidade permite a execução de comandos arbitrários, comprometendo a segurança e destruindo dados. Para mitigar o risco, é crucial validar as entradas externas e substituir `os.system()` por funções mais seguras, como `subprocess.run()`, que oferecem maior controle e segurança.

## 18 – Prompt engineering with ChatGPT

“In these files, are there any exploitable vulnerabilities? If so, how can they be exploited? Please present me the lines and the exact line numbers where the vulnerabilities are present and the path taken to exploit the vulnerability, as well as the full description of the vulnerability.” Esta foi a pergunta colocada ao chat GPT, de modo a obter uma resposta o mais completa possível.

Inicialmente, utilizámos um prompt mais simples, mas os resultados não eram muito específicos em relação ao que se pretendia, pelo que fomos aumentando a complexidade do pedido até chegarmos a esta resposta final. Testámos algumas mais complexas, mas os resultados foram de menor qualidade, revelando que tem de haver um equilíbrio entre manter o prompt o mais simples possível, mantendo a quantidade de informação que se quer saber.

Vulnerabilidades identificadas pelo programa em ambos os ficheiros.

COLE.py

Line ~52–53: Arbitrary directory creation via `os.makedirs()`.

Line ~56–58: Arbitrary file write via base64 decoding.

Line ~38–62: Lack of input validation.

LEAKS.py

Line ~38: Zip file path traversal during extraction.

Line ~79–80: Command injection via `os.system()`.

Line ~12: Insecure handling of temporary directories.

Line ~113–114: Insecure file reads and potential ReDoS vulnerabilities.

### 18.1 – COLE.py

O Chat GPT deu, de facto uma resposta correta. Esperamos que no COLE.py, a principal vulnerabilidade a ser explorada seja uma path traversal flaw, que é o que a resposta do AI indica. Graças ao prompt que fizemos, o Chat GPT explica, não só a natureza desta vulnerabilidade mas também como esta poderia ser explorada por um atacante, de forma generalizada. Indica também várias outras linhas onde ocorrem vulnerabilidades a ser exploradas pelo atacante, nomeadamente as linhas 52 e 53, onde ocorre a possibilidade de path traversal. O chat fornece uma possibilidade para explorar esta vulnerabilidade, utilizando uma linha de código simples.

De acordo com o Chat GPT existem também outras vulnerabilidades passíveis de serem exploradas neste código, designadas pelo programa como ‘Arbitrary File Write’ (linhas 56,58) e vulnerabilidade generalizada de ‘Lack of input Validation’ (linhas 38-62), pois nenhum dos inputs é verificado em termos do seu tamanho, tipo ou estrutura e torna o código vulnerável a ataques tipo Injection, Resource Exhaustion ou Malformed Data Crashes.

Neste caso, não é necessário inserir todo o código, apenas o corpo principal, sem os imports ou includes pois nenhum deles é fonte de vulnerabilidade. Caso não haja esta inclusão, não poderemos ser tão genéricos no prompt feito, uma vez que o programa vai detetar que faltam linhas no código para este funcionar corretamente e vai gastar recursos a tentar resolver também esse problema.

Para obter uma boa resposta por parte do Chat GPT, é necessário realizar iterações, conforme o necessário. Neste caso em COLE.py, não foram necessárias muitas iterações para se obter uma resposta boa em termos do que pretendíamos. Aliás, começando a pedir muitas coisas, o programa parece deixar de conseguir responder a tudo o que se pretende e as respostas tornam-se piores.

## 18.2 – LEAKS.py

No caso do LEAKS.py, a principal vulnerabilidade identificada foi uma Command Injection (linhas 79-80), do tipo OS, uma vez que utiliza o `os.system`. O chat indica-nos que utilizar o `os.system` com dados controlados pelo utilizador (jax, dex, por exemplo), permite a atacantes injetar diretamente comandos na shell. Explica ainda como aproveitar esta vulnerabilidade: Alterar jadx ou dex, com inputs tipo “; rm -rf /;” - este comando injetado executa no servidor, e pode levar a aumento de privilégios ou destruição de dados.

Há ainda mais vulnerabilidade apresentadas, designadas como Zip File Path Traversal, Improper Temporary File Handling, Insecure File Reads e Unrestricted Regular Expression Matching. De acordo com isto, durante a extração do zip file em leaks.py, não há saneamento dos caminhos, pelo que um atacante pode criar um caminho malicioso que, ao ser incluído no arquivo, pode causar alteração de ficheiros preexistentes ou colocação de outros ficheiros maliciosos em diretorias que não são supostas.

Seguidamente, indica que há a criação de diretorias temporárias com padrões previsíveis. Um atacante pode explorar esta vulnerabilidade, pré-criando a diretoria com ficheiros maliciosos, que mais tarde serão chamados no código, dado que não há verificação. O tipo Insecure File Reads tem o mesmo problema que vimos em COLE.py. Não há verificação do tipo, localização e conteúdo, que pode levar a que sejam corridos ficheiros maliciosos colocados pelo atacante nas diretorias criadas. A última vulnerabilidade apontada é Unrestricted Regular Expression Matching, que ocorre quando não há constrangimentos a expressões em input data. Segundo o Chat, isto pode ser explorado utilizando ataques ReDoS (Regular Expression Denial of Service), onde um input escolhido causa backtracking excessivo, levando a excesso de tempo a processar o input.

Neste caso, era importante fornecer os imports e includes ao Chat, uma vez que este pode determinar se as bibliotecas utilizadas são as mais seguras e sugerir outras que sejam mais robustas no que diz respeito a ataques. A biblioteca ZipFile não é a mais segura para realizar estes comandos, de acordo com o Chat GPT.

## 19 – Ask ChatGPT to fix/patch the codes

### 19.1 – COLE.py

Para este exercício fornecemos primeiramente ao ChatGPT o código e uma breve introdução para dar um pouco de contexto, de modo que ele perceba do que é que se trata. Depois disto explicamos a vulnerabilidade que o código tinha. Na primeira interação, de maneira a não encher um prompt com demasiada informação dissemos apenas isto e analisámos a sua resposta:

Prompt enviada ao ChatGPT:

Estou a desenvolver um ficheiro em python a que dei o nome COLE.py. Este código faz o seguinte:

O código COLE.py tem como principal funcionalidade o processamento de dados de pastas armazenadas em uma fila do Redis, com o objetivo de salvar esses dados em disco para que outros módulos possam processá-los



posteriormente. Ele recupera mensagens da fila COLE\_Feed\_Q, decodifica essas mensagens (que são base64 e comprimidas em gzip), e armazena-as como arquivos num diretório específico. O código também lida com a criação de diretórios se necessário e gera nomes de arquivos únicos para os dados processados. Além disso, o script regista informações sobre o andamento do processamento, como o número de pastas processadas, e pode publicar logs num canal Redis para monitorização. Este processo é contínuo, ocorrendo num loop que aguarda mensagens na fila e processa as pastas conforme são recebidas.

Em baixo segue o ficheiro desenvolvido até agora:

----- Aqui envíamos o ficheiro COLE.py -----

Analisa detalhadamente e tenta perceber bem este código.

Resposta recebida:

Devido a uma resposta muita extensa decidimos explicar mais resumidamente a resposta que o ChatGPT enviou. Basicamente foi feita uma análise extensiva ao código, com isto, a sua resposta foi dividida em 6 secções, a primeira explicava os imports do ficheiro, a segunda a função rreplace, a terceira as configurações iniciais e variáveis globais. Depois disto, dividiu a parte principal do código ( a que está dentro do ciclo while ) em 2 secções, devido à complexidade da mesma. E, por fim, na secção 6 explicou a última parte do ciclo while ( a parte onde o script remove o que é redundante e incrementa a variável que guarda o número de pastas processadas.

Com esta resposta podemos observar que fizemos um bom prompt, pois o ChatGPT agiu em conformidade com o nosso pedido (analisou o código e nada mais, focando-se a 100% no pedido feito pelo utilizador). Como a informação fornecida ao mesmo não foi muito elevada, isto fez com que o ChatGPT conseguisse analisar o código ao pormenor fornecendo uma boa explicação do mesmo (quase que linha a linha). Depois desta primeira interação, vamos agora começar por explicar a vulnerabilidade encontrada no código e, de seguida pedir para o ChatGPT tentar resolver a mesma, com a informação fornecida até ao momento.

Prompt enviada ao ChatGPT:

Este código que te foi apresentado é vulnerável a uma falha de path traversal. Quando se processa o caminho de arquivagem, na seguinte linha:

```
file_name_paste = paste.split('/')[ -1]
```

Aqui não é feita nenhuma validação ou sanitização para garantir que o caminho não contenha sequências como ../, que poderiam permitir que um atacante navegasse para diretórios fora do diretório autorizado. Como é óbvio isto é uma vulnerabilidade que pode ser explorada no meu código, o que eu quero que faças é que tentes arranjar uma solução de maneira a que esta vulnerabilidade não exista.

Resposta recebida:

Mais uma vez, devido a uma resposta muita extensa decidimos explicar mais resumidamente a resposta que o ChatGPT enviou. Nesta resposta, o mesmo começou por fazer uma introdução do problema (explicado na pergunta que recebeu). De seguida, deu uma solução que passa por verificar se o caminho para onde estamos a ir é seguro e não sai para diretórios não autorizados (diretórios que estejam fora do diretório atual). Para além disto, depois de nos fornecer com todas as alterações no código começou por explicar todas as alterações efetuadas detalhadamente, de seguida explicou os benefícios das suas alterações, ou seja, de que maneira é que com as suas alterações o código se tornava mais seguro. E, por fim, deu ainda alguns exemplos de mensagens a enviar para a fila Redis com caminhos inseguros de maneira a testar o código.

Depois desta resposta, concluímos que conseguimos ter uma interação positiva e produtiva com o ChatGPT, pois das duas vezes que interagimos com o mesmo, este respondeu sempre em conformidade com o que era suposto. Nesta última interação o mesmo ainda foi para além do que era pedido, fornecendo algumas maneiras de testarmos a sua solução. Em relação à solução em si, achamos que pode de facto solucionar o problema que descrevemos, mostrando assim que de facto descrevemos o que queríamos que o ChatGPT fizesse. Por fim, concluímos que, para uma interação produtiva com o ChatGPT é necessário descrevermos bem o nosso problema (explicando pontos mais complicados de perceber se possível) e fornecer o mínimo de informação em cada uma das prompts, de maneira a existir uma comunicação positiva e com o mínimo de complexidade possível.

## 19.2 – LEAKS.py

No caso de LEAKS.py, fornecemos o ficheiro LEAKS.py ao Chat GPT e pedimos-lhe que analisasse o código, tendo em conta aquilo que é o objetivo deste código e, posteriormente, para que corrigisse a OS Command Injection Flaw nele presente, sem fazer alterações significativas no código. Utilizámos o seguinte prompt:

“O objetivo do arquivo LEAKS.py é identificar e extrair informações sensíveis ou vazamentos de dados em arquivos APK (Android Package Files). Ele faz isso ao realizar uma análise detalhada dos arquivos APK, descompilando-os e aplicando padrões de busca para identificar possíveis dados expostos ou inseguros. Analisa o código que enviei e corrige a vulnerabilidade de OS Command Injection Flaw nele presente, sem comprometer o restante código.”

O chat respondeu o seguinte:

O código contém uma vulnerabilidade de OS Command Injection no método decompile, devido ao uso de os.system com comandos concatenados. Para corrigir isso sem comprometer a funcionalidade do restante do código, substituiremos os.system por uma abordagem mais segura usando a biblioteca subprocess, que permite passar argumentos diretamente sem expô-los a injeções de comandos.

--- (o chat em seguida apresenta as alterações ao código, demasiado extensas para colocar tudo aqui, colocaremos apenas as linhas pertinentes) ---

Segue a versão corrigida do método decompile:

```
import subprocess # Import necessário
def decompile(self):
    self.writeln("*** Decompiling APK...", clr.OKBLUE)
    with ZipFile(self.file) as zipped:
        try:
            dex = os.path.join(self.tempdir, f"{self.apk.package}.dex")
            with open(dex, "wb") as classes:
                classes.write(zipped.read("classes.dex"))
        except Exception as e:
            sys.exit(self.writeln(str(e), clr.WARNING))

# Usando subprocess para evitar injeções de comando
args = [
    self.jadx, # Path para o jadx
    dex,       # Arquivo dex gerado
    "-d",      # Diretiva de saída
    self.tempdir,
    "--deobf"  # Argumento adicional para desobfuscação
]

try:
    subprocess.run(args, check=True)
except subprocess.CalledProcessError as e:
    sys.exit(self.writeln(f"Error during decompilation: {str(e)}", clr.WARNING))

return self.tempdir
```

Em seguida, o Chat GPT explica as alterações propostas e o motivo de as ter feito.

“Substituição de `os.system` por `subprocess.run`:

- `subprocess.run` aceita uma lista de argumentos, onde cada elemento é tratado separadamente, eliminando a necessidade de escape manual e prevenindo injeções.
- O parâmetro `check=True` assegura que erros de execução sejam tratados de maneira explícita.

Manutenção da Estrutura Original:

- O método preserva o comportamento original, mas agora é imune a comandos maliciosos injetados via argumentos como `dex` ou `jadx`.

Validação Automática:

- Qualquer falha durante a execução do comando (como um binário `jadx` inválido) resulta em uma exceção que é capturada e tratada, garantindo robustez.

Teste de Segurança:

Com a correção implementada, mesmo que argumentos como `self.jadx` ou `dex` sejam manipulados para incluir sequências maliciosas (`"; rm -rf /;"`), eles serão tratados como strings literais e não executarão comandos adicionais.”

A solução proposta e descrita pelo Chat GPT adequa-se ao problema em questão, mantendo o propósito original do programa e, em grande parte, a sua estrutura, uma vez que apenas o método `decompile` foi alterado. Foi importante especificar a parte do código a corrigir, uma vez que sabemos que há mais vulnerabilidades possivelmente no `LEAKS.py` e, sem essa especificação, o programa tentaria corrigir tudo ao mesmo tempo, possivelmente cometendo erros e sobrepondo comandos. Deste modo, conseguimos evitar isso e obter uma resposta clara e concisa, fácil de aplicar diretamente no código base.