Em primeiro lugar, é importante ter em mente que o sistema operacional Linux favorece a resolução de todos os desafios, de qualquer competição. O motivo disso é que o terminal do Linux oferece uma gama de comandos/ferramentas simples e bem diretas para facilitar grande parte do trabalho de procura por algo. Em uma competição, a procura é pela flag (resposta) e esta resposta, nos CTFs lineares não costumam ter um formato definido (como "CTF{flag_aqui}"). Podem ser apenas a string "flag".

Mostraremos a resolução dos desafios utilizando primariamente o Windows e, alternativamente, o Kali Linux (apenas para referência, utilize qualquer Linux).

Desafio 1: Payload Ofuscado.c

Link: xmas.sanjose.institute/ctf/n/1-429202/desafio_payload_ofuscado.c

1ª Etapa: Ao abrir o site, ele nos dá um arquivo "desafio_payload_ofuscado.c". No início do arquivo, existe um enunciado pedindo para que a pessoa encontre um 'payload' dentro do arquivo e exige 'olhos de lince', o que indica que existe algo no conteúdo do arquivo que será visível/legível ou que pelo menos saltará aos olhos. Lendo o conteúdo, começa uma sequência de logs iniciados pela frase "COMEÇO DO RUÍDO", explicitando que trata-se apenas de ruído, não há nada interessante ali. Rolando para baixo, aparece outra seção do código dizendo ser um fragmento de hexadecimais. Rolando mais para baixo, se chega no 'payload cache' que parece bater com a proposta informada no enunciado, então focaremos mais nesta parte do código. Ela não é tão grande, facilmente salta aos olhos a sequência de letras 'zip' no código. Lendo um pouco antes do 'zip', dá para se identificar a string "vcme3ncontrouzip". Ou seja, a próxima etapa está em "vcme3ncontrou.zip"

2ª Etapa: O arquivo zipado contém um README.txt e um app.js. Abrindo o README.txt, o enunciado diz que quer ver se é possível encontrar qual é a falha da seguinte aplicação do app.js. Ao abrir o app.js, trata-se de uma aplicação de guestbook. No código, é possível ver um trecho de um código de escape em JavaScript, que normalmente é utilizado para ataques de XSS. No comentário do código, existe a frase "parece que está tudo bem com XSS". Neste contexto, é importante destacar que a primeira análise que pode ser feita (para simplificar) é se existe falha de XSS no código ou não, para descartar esta possibilidade. Se sim, essa é a resposta. Se não, foi algo para despistar. Logo em seguida no código, é possível ver a função 'addGuestbookEntry' que efetivamente adiciona uma entrada no guestbook (como se fosse um post em um fórum, é o mesmo mecanismo), nesta função, apesar de estar comentado no código que esta parte está no modo "seguro/safe" e vários detalhes que fazem o código parecer seguro, no trecho 'messageHtml' é possível perceber que o que o usuário/visitante do site digita é publicado em HTML! Ou seja, um texto simples é convertido em texto enriquecido em HTML, o que indica que, caso o usuário queira submeter um código HTML, o site vai renderizar e exibir na página o código HTML que o usuário quiser. Isso indica ataque de XSS claro. Além disso, na primeira linha do app.js, é dito que a sigla da falha abreviada (XSS, sigla de Cross-Site Scripting) seguida de ".txt" é a próxima etapa. Ou seja, XSS.txt.

3ª Etapa: Na parte final, é possível ver uma trivia, uma pergunta de conhecimento sobre XSS, é perguntado qual tipo de problema os Trusted Types resolvem e indicam que apenas a sigla abreviada seguida de .txt é o próximo arquivo. Ao pesquisar Trusted Types XSS no Google, é possível observar a resposta clara de que Trusted Types é uma API segura para prevenir um tipo específico de XSS, o XSS em DOM. Próximo arquivo é DOM.txt. O DOM.txt revela a flag: "d0mhackerr".

Flag final: d0mhackerr

Desafio 2: HQs.jpg

Link: xmas.sanjose.institute/ctf/n/2-184905/HQs.jpg

1ª Etapa: Trata-se de um arquivo de imagem, que, ao abrir, percebe-se ser uma imagem bem grande. Quando um desafio apresenta uma imagem grande (e os desafios estão no nível fácil), é muito possível que tenha algo escondido de forma visível na própria imagem. Então, é possível dar um zoom na imagem e analisar a olho nu. Logo a primeira vista já é possível identificar que, na própria imagem, está escrito "barcode.gif".

2ª Etapa: Ao abrir o arquivo, percebe-se um longo código de barras. Existem vários tipos de códigos de barras, o mais comum e utilizado de todos parece ser este que foi usado, chamado de Code 39. Abrindo um leitor qualquer de código de barras, já é possível converter a imagem em seu texto puro, o texto encontrado é "vcemtcuriosopodeirprabompontozip" (ou, "vc e mt curioso pode ir pra bom.zip"), caso gerasse alguma dúvida, era possível testar também "bomponto.zip", algo que podia confundir.

3ª Etapa: Ao acessar bom.zip, é baixado um arquivo zipado que contém uma pasta chamada bomzip e com várias outras pastas dentro. Abrindo as primeiras pastas internas e os primeiros arquivos, é possível perceber que há muito ruído, muita coisa sem sentido colocada apenas para poluir, incluindo diversos arquivos .zip corrompidos. Algumas pastas, ao abrir, você já se deparava com arquivos. Outras, ao abrir, você se deparava com mais pastas. Navegando entre as pastas, somente um arquivo salta aos olhos, na pasta "src". Na pasta "src", haviam mais pastas, como a "c", a "core", a "js", a "python" e a "tools". Em todas elas aparentemente o conteúdo eram mais arquivos de ruído. Se este desafio fosse bastante complicado, talvez a flag ou uma pista estaria escondida dentro de alguns dos arquivos de ruído, mas, por se tratar de um desafio de nível fácil ou médio, é esperado que se encontre algo mais trivial e, de fato, na pasta "tools", o conteúdo eram dois arquivos, um arquivo em python e um arquivo zip chamado "files.zip" que também estava corrompido, mas o nome dele é totalmente diferente dos demais, o nome em si não é um ruído, e o tamanho dele também gera suspeitas, dado que ele tinha apenas 1KB, sendo o único arquivo zip deste tamanho entre todas as pastas. Como ele está corrompido, a primeira técnica que deve ser utilizada é entender se o arquivo de fato é um zip, para isso, você pode utilizar o comando 'file {nomedoarquivo}' para verificar. Outra opção, é você abrir em um editor de hexadecimal para checar nos primeiros bytes do arquivo do que se trata. Logo ao abrir em um editor de hexadecimal (ou no próprio bloco de notas) já é possível encontrar uma frase cifrada. A frase é 'zg qi zirgiy e jpek jmrep il gxjgsqxigrmgewhigsqtyxeges', e parece ser uma cifra de substituição/transposição. Utilizando a ferramenta dcode.fr, podemos fazer um brute-force no deslocamento, caso seja Cifra de César. Com esta ferramenta, se descobre o texto puro e o deslocamento sendo 4. O texto puro é: 'vc me venceu a flag final eh ctfcomtecnicasdecomputacao'.

Flag final: ctfcomtecnicasdecomputacao

Desafio 3: musicalidade-longa.txt

Link: https://xmas.sanjose.institute/ctf/n/3-844230/musicalidade-longa.txt

1ª Etapa: O txt começa com um enunciado dizendo que você deverá encontrar novamente uma "agulha no palheiro" e que é possível resolver sem técnicas, mas que com técnicas será mais fácil. Como o desafio aparenta ser só isso, algo está escondido no txt. De imediato é possível perceber que a barra de rolagem da página está ativa, ou seja, que o txt é mais longo do que cabe na tela. Arrastando para baixo e chegando por volta da metade do txt, é possível encontrar a string '6ODycFBH2yw youtube', que indica que há algo nesse endereço do YouTube. Para acessar o vídeo, basta entrar em youtube.com/watch?v=6ODycFBH2yw.

2ª Etapa: Abrindo o vídeo, nota-se que o mesmo tem quase 6 horas de duração. Nestes casos, procura-se algo no áudio ou no vídeo/imagem. Ao reproduzir o vídeo, é possível notar que músicas tocam durante todo o vídeo. Colocando o vídeo em um editor de vídeo para visualizar as ondas do áudio, é possível ver que não há intervalos notáveis, nem mudanças que saltem aos olhos. Então, podemos deixar o áudio em segundo plano (e voltar para ele apenas se não encontrarmos nada no vídeo/imagem) e partir para analisar o que aparece no vídeo. Arrastando o mouse por todo o vídeo, de pouco em pouco, é possível ver as informações úteis para a próxima parte. Para resolver de maneira mais fácil, o ideal seria utilizar uma ferramenta para separar todos os quadros do vídeo em imagens, removendo as duplicadas e checando apenas as que destoam/mudanças abruptas. Na minutagem 01:01:13 é possível ver 110.zip escrito na parte inferior do vídeo de forma bem grande.

3ª Etapa: Abrindo o 110.zip, é possível ver que há dois arquivos de texto, um chamado README e outro chamado tcpdump. Ao tentar extrair para visualizá-los, é pedida uma senha. A primeira tentativa, nesses casos onde o desafio não nos deu pista da senha, é de testar as mais comuns, 1234, 123456, 12345678, admin, administrator, administrador, password, password123, e mais algumas 3 ou 4 opções de senhas mais utilizadas. Não funcionando, tentaria algo com relação ao nome do arquivo, tentaria 110, 110.zip, 110zip ou 6, pois 110 em binário representa o número 6. Ao não funcionar nenhuma dessas opções, podemos tentar um brute-force local (que não é contra as regras, pois o arquivo está localmente no computador, não tem interação nenhuma com a infraestrutura do CTF) de senhas. É possível que mesmo assim não funcione (alguns dicionários podem acertar a senha, mas são poucos!), neste caso, o desafio ainda deve esconder alguma coisa. Voltando para o vídeo e continuando a pairar o mouse pelo vídeo até o final, muito próximo dos 2 minutos finais do vídeo, em 05:54:11, aparece no canto inferior direito em azul a senha: 'admin1234'.

4ª Etapa: Com os arquivos do zip extraídos, abrimos o README para verificar o que pede o desafio. É dito que um comentário bem suspeito foi feito em uma publicação. Isto e um tcpdump indica que precisamos verificar o tráfego de rede para encontrar uma publicação, um comentário ou algo de rede social. A maior parte dos pacotes não tem qualquer string em texto puro, então isso facilita bastante em navegar pelo arquivo até encontrar um ponto de interesse. Após navegar pelo 1/4 inicial do arquivo, é possível ver a primeira menção a uma rede social, o pacote para Instagram.com. Logo em seguida deste pacote, uma requisição GET para

www.instagram.com/sanjose.institute. Dois pacotes à frente, também aparece um link exato que foi acessado: instagram.com/sanjose.institute/p/CzP0sJqL9Aa/. Ao acessar, a página não existe. No formato padrão do Instagram, os posts não são identificados pelo autor, então, o link deveria ser instagram.com/p/CzP0sJqL9Aa/ mesmo se for um post de @sanjose.institute. Ao acessar, também não existe. Voltando ao tcpdump, poucos pacotes adiante é possível visualizar uma string solta no meio de um pacote de post de comentário em uma foto do Instagram dizendo 'findme!', ao tentar isto como flag final (com e sem exclamação) não se obtém sucesso. O resto dos identificadores de publicações e comentários não parecem ser válidos e o resto do tcpdump não mostra mais nada interessante. Voltando ao README, o enunciado contém o termo OSINT no começo, que indica que é um desafio de OSINT, ou, neste caso, SOCMINT, não se trata de procurar algo no tcpdump e sim nas redes sociais (inteligência em fontes abertas/públicas, que incluem redes sociais abertas). Para seguir no mesmo caminho, decidimos por buscar algum comentário suspeito feito em publicação do perfil @sanjose.institute. Não são muitas publicações, então não será tão difícil encontrar. Atualmente, é possível passar o mouse em cima de cada publicação na timeline para ver quantos comentários existem naquela publicação. As primeiras 28 publicações quase não tem comentários, então rolamos logo para baixo, nas primeiras publicações para analisar a partir dali. Na publicação de 15 de janeiro de 2023 é possível ver um comentário dizendo 'lescl.txt'.

5ª Etapa: Ao abrir o 'lescl.txt', existem várias sequências de binários separadas por quebra de linha e ao final com uma sequência maior. Decifrando a primeira sequência, já se tem o resultado 'm', então, pegamos um trecho mais longo de sequências (as primeiras 14 sequências) e tentamos decodificar para ver se o texto puro continua aparecendo de forma direta. Aparece um trecho dizendo que é preciso decodificar algo. Logo, decidimos por pegar todo o binário, inclusive a sequência mais longa ao final e decodificar de uma vez só. O resultado é que uma frase dizendo que será necessário decodificar algo mais difícil e a string 'c93ccd78b2076528346216b3b2f701e6', pelo comprimento e formato da string, é possível dizer que com segurança que, caso seja uma criptografia, se trata de uma função de hash de 128 bits (por conta dos 32 caracteres em hexadecimal - cada dígito hexadecimal representa exatamente 4 bits, 32 x 4 = 128). Dado isso, só pode ser MD5, MD4, MD2 ou RIPEMD-128. Sendo a mais comum, o MD5. Utilizando a ferramenta online CrackStation, que já testaria MD5, MD4 e MD2 ao mesmo tempo, além de conter o banco de dados mais vasto de correspondências do MD5, se tem o resultado: admin1234. Mas esta ainda não é a flag final! Então, é o próximo passo, podemos tentar as extensões mais pedidas até agora, sendo a mais pedida o .zip.

6ª Etapa: Ao acessar admin1234.zip, o arquivo compactado é baixado e contém 2 arquivos, um README em texto e um vídeo MP4. No README, o desafio informa que a etapa final está no arquivo que tem o nome do que está sendo construído no vídeo .txt em maiúsculo. Ao abrir o vídeo, aparecem vários profissionais de capacete em uma obra, no meio de bastante terra batida, além disso, é exibida uma marca d'água da página que publicou o vídeo, "Grupo Kaza". Existem duas formas de abordar este desafio: com técnicas de GEOINT (a mesma utilizada no famoso jogo Geoguessr) para descoberta de geolocalização apenas com base no que é visto no vídeo e com SOCMINT para tentar descobrir a fonte do vídeo, que o nome do arquivo denuncia ter sido baixado do Instagram e a marca d'água indica que provavelmente o vídeo pode ser encontrado no Instagram do Grupo Kaza. Resolução via SOCMINT: no perfil grupo.kaza do Instagram, rolando o feed, é possível encontrar o vídeo publicado em 30 de junho de 2025, com a legenda dizendo que

aquele local será o "Parque Una". Como é necessário uma palavra e em maiúsculo, o certo seria UNA.txt. Resolução via GEOINT (inferências geográficas): no segundo 00:13 do vídeo, é possível ver uma estrutura bem grande ao fundo, que pode indicar um excelente ponto de interesse. No início do vídeo, é mostrada a área com visão aérea que demonstra ser uma cidade com um grande espaço para construção, de frente para um bairro bem estruturado, com muitos prédios, uma longa avenida e o que parece ser um estabelecimento, seja centro comercial ou shopping. Alguns segundos depois, é possível ver as lixeiras coloridas, os carros e suas placas, indicando ser um local no Brasil. No segundo seguinte, é possível ver um vasto horizonte de mata, com o que aparenta ser muitos e muitos quilômetros em um ângulo considerável de apenas verde. Isso indica fortemente ser uma cidade de interior. Pesquisando o Grupo Kaza, é possível ver que eles são uma imobiliária presente em São Paulo, São José dos Campos e Jacareí. Muito mais presente em São José dos Campos. É possível descartar São Paulo ou tornar esta uma opção secundária para validar alguma cidade de interior primeiro. Ao abrir a visão via Satélite com relevo no Google Earth, é possível perceber que Jacareí não tem aquele local mostrado na imagem, com uma avenida de frente para uma vasta área vazia para construção e de frente para muitas dezenas de prédios, um centro comercial e um bairro lateral. Ainda no Google Earth, passando para São José dos Campos, ao buscar em alto nível pelo mesmo formato de terreno e mesmos pontos de interesse, é possível identificar facilmente na região central o terreno com terra em formato que lembra de um peixe, no Jardim Alvorada. Abrindo a mesma localização no Google Maps, o nome do terreno vazio para construção aparece: 'Parque UNA São José dos Campos'. UNA.txt seria então o próximo passo.

Flag final: parqueegames

Desafio 4: Enigma das Portas

Link: xmas.sanjose.institute/ctf/n/4-012936/enigma_ctf

1ª Etapa: A primeira coisa curiosa é que não foi mostrado uma extensão, desta vez. Isto já pode indicar algumas coisas, pode indicar se tratar de uma aplicação web (/enigma_ctf/index.html) ou um binário executável (para Linux, pois para Windows, haveria o .exe). Ao abrir, é possível ver uma explicação sobre ele ter sido feito para Linux, mas tendo problemas de compatibilidade com Arch e MacOS, também inclui uma referência para o arquivo "enigma_ctf_pack.zip", para se ter mais informações. Como mais nada foi mostrado, entende-se que o desafio continua lá. Então, acessando enigma_ctf_pack.zip, é baixado um zip com 3 arquivos, o arquivo 'README_BINÁRIOS', o arquivo 'enigma_ctf_linux_x86_64' e o arquivo 'enigma_ctf_linux_x86_6_static'. Agora fica claro que ambos são arquivos executáveis do Linux. O conteúdo do README é basicamente uma documentação do que deve ser feito para rodar o executável sem problemas de compatibilidade, recomenda-se executar no terminal o arquivo 'enigma_ctf_linux_x86_64' por ser dinâmico. Caso dê erro, indica que faltam bibliotecas essenciais para rodar o arquivo, então recomenda-se utilizar a versão estática (ou a dinâmica em outro sistema!) 'enigma_ctf_linux_x86_64_static'. No Windows, através do WSL, é possível rodar com tranquilidade. No Kali Linux, é possível rodar com tranquilidade, inclusive o executável dinâmico! Para a resolução deste desafio, utilizaremos o WSL. Começando por dar chmod +x para o executável e executando-o através do ./enigma_ctf_linux_x86_64. Ao executar, já aparece no terminal as instruções que mostram que se trata de um jogo de perguntas e respostas que pune chutes e que tem tempo para responder, tornando mais curto o tempo a cada resposta. A maior dificuldade está na punição pelo erro, ao errar, você volta do início e são 15 perguntas. Para a primeira pergunta, são 45 segundos para a resposta, para a 15ª, apenas 15 segundos.
Cabe destacar aqui duas resoluções possíveis: você pode tentar jogar o jogo ou hackear o jogo. Ou seja, você pode responder as perguntas e, com isso, conseguir a flag (pode ser estressante, pode ser demorado, pode ser chato, pode dar trabalho...) ou pode descobrir a falha do jogo com debug do executável e extração da flag de dentro do binário (pode dar trabalho).

As perguntas/enigmas do quiz (as respostas estarão mais adiante):

1. “O arquivo de usuários não guarda o segredo. O segredo mora no irmão mais restrito (hashes de senha). Pergunta: qual é o caminho absoluto desse arquivo?”
2. “Três blocos de 8 bits: 01000011 01010100 01000110. Interprete como ASCII. Pergunta: qual palavra de 3 letras aparece?”
3. “Permissões em octal: dono: rwx, grupo: r-x, outros: ---. Pergunta: qual é o modo octal (3 dígitos)?”
4. “Redes: quantos endereços TOTAIS existem numa rede IPv4 /27? Pergunta: responda só o número.”
5. “IPv4 sem opções: tamanho mínimo do cabeçalho. Pergunta: quantos BYTES?”
6. “SHA-256 gera 256 bits. Em hexadecimal, isso vira um tamanho fixo. Pergunta: quantos caracteres tem o digest em hex?”
7. “Existe um sinal POSIX que encerra um processo sem apelação. Não pode ser capturado nem ignorado. Pergunta: nome do sinal (ex: SIGKILL).”

8. "Uma trilha de HEX foi achada num bilhete: 2f6574632f706173737764. Leia como bytes ASCII. Pergunta: qual é o caminho absoluto resultante?"
9. “DNS: qual registro (uma única letra) aponta nomes para IPv4? Pergunta: qual é a letra?”
10. “No /proc existe um arquivo que lista estatísticas das interfaces (RX/TX). Pergunta: qual é o caminho completo?”
11. “Um byte x: x = 10110110 (binário). Faça: x >> 3. Pergunta: resultado em binário com 8 bits (com zeros à esquerda)?”
12. “Um comando foi escrito em BINÁRIO (ASCII, 8 bits por letra): 01100011 01101000 01101101 01101111 01100100. Pergunta: decodifique e responda o comando (apenas a palavra).”
13. “Quatro bytes apareceram num dump (hex): 78 56 34 12. Interpretados como inteiro 32-bit little-endian, viram outro hex. Pergunta: qual é o valor final em hex (sem 0x, minúsculo)?”
14. “Regex: qual símbolo ancora a busca no INÍCIO da linha? Pergunta: responda com um único caractere.”
15. "Sou o único número que veste a capa do superusuário no Linux. Pergunta: qual é o UID (apenas o número)?"

Resolução 1 (Jogando o Jogo): Ao executar o programa no terminal, é pedida uma seed para gerar seu identificador de jogador e, em seguida, a primeira pergunta já aparece. As perguntas são de tecnologia, especialmente de Redes e Sistemas Operacionais. Considerando que é praticamente impossível responder a todas as perguntas sem nenhum tipo de pesquisa ou erro, você terá que voltar ao início muitas vezes, pois, se uma pergunta é respondida incorretamente, o quiz volta para a primeira pergunta. Sucessivas falhas levam a uma suspensão de alguns segundos, te impedindo de continuar o jogo/continuar a responder. Ou seja, é algo mais feito para atrasar. Dado esse cenário, você precisará: responder as perguntas corretamente (parte mais fácil) e evitar errar sucessivas vezes. Para responder as perguntas corretamente o mais rápido possível, você "precisa" errar, pois vai estourar inevitavelmente o tempo limite para responder a maioria das questões. Neste caso, a estratégia que mais faz sentido é:

1. Anotar a pergunta que aparecer no bloco de notas.
2. Responder ou pesquisar para encontrar a resposta e anotá-la no bloco de notas.
3. Colocar a resposta no programa e voltar ao passo 1.

Dessa forma, você garante que vai primeiro passar por todas as perguntas, responder cada uma individualmente e anotar as respostas corretas. No fim, em posse de todas as respostas corretas, apenas passa por todas copiando e colando a resposta do bloco de notas e finalizando o desafio.

Resolução 2 (Hackeando o Jogo): Para quem se sente confortável com debugging e engenharia reversa, ou que queria tentar não ter que responder todas as perguntas, para ver se seria mais rápido, é possível burlar a competição ao conseguir extrair a flag final de dentro do programa sem ter que responder corretamente todas as perguntas. Inclusive, durante o processo, é possível também "hackear" as perguntas para aceitarem qualquer resposta como resposta correta e te aprovar de pergunta em pergunta até chegar no final e soltar a flag final do desafio. É de máxima importância que o conhecimento a seguir seja utilizado SOMENTE para fins didáticos em ambiente de treino ou controlado, que você possua permissão expressa para burlar.

- **Método para todas as respostas serem corretas independentemente do input:** Muitos programas checam a resposta com strcmp, strncmp, memcmp, etc. Nós podemos "enganar" o programa carregando uma biblioteca nossa antes (a LD_PRELOAD) que substitui essas funções e sempre diz "é igual". Primeiro passo, no WSL/Kali, crie uma pasta e entre nela:
  $~ mkdir -p ~/ctf_cheat && cd ~/ctf_cheat
  Agora crie o arquivo:
  $~ nano cheat.c
  Salve o código no nano:
  #define _GNU_SOURCE
  #include <string.h>

  int strcmp(const char *a, const char *b) { (void)a; (void)b; return 0; }
  int strncmp(const char *a, const char *b, size_t n) { (void)a; (void)b; (void)n; return 0; }
  int memcmp(const void *a, const void *b, size_t n) { (void)a; (void)b; (void)n; return 0; }

  Compile a biblioteca cheat.so:
  $~ gcc -shared -fPIC -O2 -o cheat.so cheat.c

  Se gcc não existir (o que é comum no WSL):
  $~ sudo apt update && sudo apt install -y build-essential

  Para Arch Linux:
  $~ sudo pacman -S --needed base-devel

  Rode o binário com o LD_PRELOAD, para isso, vá até a pasta onde está o executável do desafio e rode assim:

  LD_PRELOAD=$HOME/ctf_cheat/cheat.so ./enigma_ctf_linux_x86_64_static

  Agora você pode digitar qualquer coisa nas perguntas e ele tende a aceitar tudo e no final imprime a flag. Disclaimer: Este método só funciona em alguns sistemas e na versão dinâmica do executável.
- **Método para ver a flag passando (mesmo sem entender assembly):** Aqui a ideia é: mesmo que o programa esconda a flag, ele precisa montar a flag em memória e imprimir no final, aí você vai farejar a impressão. Utilizando técnicas de debugging, podemos pegar as instruções de assembly do binário, localizar o trecho do código que imprime a flag, colocar um breakpoint ali, ou "pular" para lá (jump) ou mesmo forçar o caminho (alterando o registrador ou o condicional). Dessa forma, conseguimos pedir para o programa nos dizer o que ele mostra ao final, isto é, a flag.
  A ferramenta que utilizaremos para o debug será o GDB, ele executa o programa e interpreta cada linha do assembly do programa compilado, permitindo que a gente pause a execução, veja registradores, mude um fluxo (jump que mencionei anteriormente) e veja onde funções são chamadas, por exemplo.
  Para rodá-lo e no modo silencioso (quiet mode) para deixar a tela menos poluída, rodaremos o comando:

$~ gdb -q ./enigma_ctf
Uma vez aberta a interface, mandamos o seguinte comando dentro do gdb:
(gdb) set pagination off
Dessa forma, não precisaremos ficar respondendo as solicitações do gdb, aí não precisaremos ficar apertando ENTER infinitamente. Com o set pagination off, o GDB vai imprimir tudo direto na tela. Em seguida, enviamos o seguinte comando:
(gdb) catch syscall write
Que significa basicamente "pare" (catchpoint) quando acontecer um evento (exceções, syscalls, signals, etc). Ou seja, existe um pause toda vez que o processo fizer a syscall write. Em Linux, o write é a syscall que escreve bytes em um arquivo. Ou seja, quase tudo que o programa imprime no terminal/na tela passa por um write (direto ou por funções como printf/puts, que por baixo acabam chamando o write). Fazemos isso porque o intuito ali é "assistir" o programa imprimindo coisas (chamando o write) até que ele imprima o que a gente quer, a flag. Os próximos dois comandos dentro do gdb são:
(gdb) condition 1 $rdi==1
(gdb) run
Fazendo isso, a gente aplica uma condição ao catchpoint. Como só criamos um catchpoint, ele é o 1. E a expressão $rdi==1 só diz que essa condição precisa ser verdadeira para parar. Sem essa condição, o GDB vai parar em todas as escritas, ficando insustentável visualizar tudo. Com o run, o programa é executado dentro do gdb. Quando o gdb parar, você verá algo como fd=1 ou buf=0x... ou até nbytes=.... Nós queremos ver o que está dentro de buf, então, utilizamos o comando:
(gdb) x/s $rsi
(gdb) bt
O x é para examinar a memória, o /s é para tratar como string em C e o $rsi é o ponteiro do buffer (buf) que está sendo escrito. Então, x/s $rsi mostra o texto que o programa está tentando imprimir. Já o bt, mostra a stack/pilha de chamadas, que é o que nos vai dar exatamente o ONDE que as coisas estão acontecendo (em hexadecimal).
O output do bt vai mostrar que temos a stack chegando no main() (provavelmente no frame #6). O próximo passo é achar, no assembly do main, o ponto que representa o que queremos, sem ter que chutar offset (chutar possíveis hexadecimais pra tentar acertar a localização do assembly que solta a flag).
Digite a seguinte sequência de comandos, ainda dentro do gdb:
(gdb) frame 6
(gdb) set logging file main_disasm.txt
(gdb) set logging overwrite on
(gdb) set logging on
(gdb) disassemble main
(gdb) set logging off
Explicando de forma breve, isso seleciona o frame que queremos investigar, o frame é o nível da stack de chamadas, cada função chamada cria um "frame". Queremos ir para o frame onde o programa chamou o write! (__libc_write). Em seguida, definimos para qual arquivo o GDB vai escrever tudo que ele imprimir quando o logging estiver ligado (main_disasm.txt) e depois ainda ativamos o overwrite, para zerar um arquivo se já existir e substituir ele pelo conteúdo que tivermos agora. Depois disso é ligado o logging, ou seja,

tudo que o GDB imprimir no console agora vai para esse txt. Aí começa a investigação de fato, o disassemble é para o GDB mostrar o assembly exato da função main (quando não está ofuscado, é simples assim!), dessa forma, conseguimos ver blocos importantes e anotar os offsets (que é tipo as coordenadas de ONDE o programa está fazendo cada coisa, tipo "main+0x66e", no assembly de main na "linha" 0x66e, ou seja, na linha 1646). Feito isso, é desligado o logging, pra não precisar mais gravar informações no arquivo. Feito isso, vamos investigar o arquivo com o seguinte comando (pode rodar no gdb, mas se não funcionar, dê "quit" e rode no terminal normal sem o "shell"):
(gdb) shell grep -n "0xf" main_disasm.txt | head -n 10
ou
$~ grep -n "0xf" main_disasm.txt | head -n 10
A ideia aqui é a seguinte: o quiz tem 15 perguntas/enigmas, então, 0xf = 15, ou seja, é um marcador do desafio (total de fases), então, procurar por ele é uma forma bem didática de encontrar onde o programa decide que você venceu. Repare que no dump também aparece "%r15d,%r15d", isso zera o contador, outro forte indício de que r15d é "progresso/acertos". Naquele bloco do output, o 'cmp $0xf,%r15d' checa se já chegamos em 15, o 'je main+1646' é um "se sim, pula pra vitória" e no main+1646 aparecem vários:
lea ...,%rdi
call puts@plt
...
lea ...,rdi
lea ...,rsi
call printf@plt
O que significa tudo isso? É basicamente o compilador colocando dentro de rdi o endereço de uma string que está na seção "somente leitura" (que é o tipo .rodata do binário). Logo em seguida vem o call puts@plt, ou seja, ele está fazendo um 'mostrar("alguma string"). Por que antes colocamos que o printf da flag fica em $rsi? Bem, é porque essa é a convenção padrão de chamadas no Linux x86_64 (System V AMD64 ABI): o 1º argumento de função é o rdi, o 2º argumento é o rsi, o 3º argumento é o rdx e o 4º é o rcx (e por assim em diante). Logo, isso aqui:
lea ..., %rsi ; "alguma string"
lea ..., %rdi ; "FLAG: %s\n"
xor %eax, %eax; detalhe típico antes de printf (variadic)
call printf@plt
Ou seja, isso tudo é exatamente um: 'printf("FLAG: %s\n", alguma_string)'.
Como a gente já achou a pista mais importante, que é esse cmp $0xf,%r15d (0xf = 15), porque tudo indica que é o "checador" de quantas perguntas foram acertadas, a gente precisa ver o bloco logo após ele pra identificar qual ramo é o "vitória/flag". Rode:
(gdb) shell nl -ba main_disasm.txt | sed -n '290,330p'
Aqui estamos buscando ver as instruções em torno de 0x...57ae <main+1646> e se aparecem puts/printf e a parte que imprime a flag!
Com o output, você já conseguirá ver claramente o bloco de "vitória": 'cmp $0xf,%r15d' (linha 302) significa "acertou 15?", 'je ... <main+1646> (linha 303) significa "se sim, vá pro bloco de vitória" e em main+1646 (linha 352 em diante) ele começa a imprimir mensagens e imprime a flag. Repare exatamente no trecho entre as linhas 356 e 357: 'lea ... %rdi #

0x555555556800' e 'call puts@plt'. Ele está te dizendo que chama o 'puts' do C para imprimir a string que está no endereço 0x555555556800! E você já viu este endereço antes sendo um buffer impresso, lembra? Então, para recapitular de forma resumida, a receita até aqui é simples de entender, primeiro passo é achar o contador de acertos (procurando por cmp $0xf, <registrador>), o segundo passo é seguir o salto condicional (ver o destino do 'je' ou 'jne', etc), o terceiro passo é ver o que o bloco imprime (no bloco destino, procurar por call puts/printf) e o quarto passo é mostrar a string que está sendo impressa (inspecionar o endereço que vai pra 'puts', o '%rdi').
Agora vamos para a reta final, já temos o endereço do bloco e o endereço do buffer que o 'puts' imprime. Para fazer do jeito mais simples, é só colocar um breakpoint (um delimitador no programa) no bloco de vitória e rodar:
(gdb) b *main+1646
(gdb) run
Quando parar no breakpoint, rode:
(gdb) x/s 0x555555556800
Isso vai mostrar a string que o programa vai imprimir (que provavelmente é a flag). Se aparecer só "======", a flag não está nesse endereço ainda, aí a flag é a string usada no printf da linha 361, e vamos inspecionar os ponteiros em %rdi e %rsi.
Se 'x/s 0x555555556800' não mostrar a flag, então rode também:
(gdb) x/s $rdi
(gdb) x/s $rsi
Se não aparecer a flag, é porque o catchpoint lá do write ainda está travando as inspeções, então temos que parar no ponto do main que chama o printf da vitória, porque nesse printf o 1º argumento é o format e o 2º argumento é a string da flag.
Então, desabilite o catch do write (senão você vai ficar 300 vezes no print):
(gdb) info breakpoints
(gdb) disable 1
(Assumindo que o catchpoint lá atrás era o breakpoint 1, se não for, desabilite o número certo que aparecer)
Em seguida, coloque breakpoint no printf do bloco de vitória (você já viu no disasm que é main+1698):
(gdb) b *main+1698
Aí você pula pro início do bloco de vitória (que você descobriu ali em cima seguindo o 'je' depois do 'cmp $0xf,...)':
(gdb) jump *main+1646
(gdb) continue
Quando parar no breakpoint do printf, rode:
(gdb) x/s $rdi # format da string
(gdb) x/s $rsi # argumento 1 (normalmente a flag)
A flag deve estar em $rsi nesse ponto.
Se $rsi não for a flag (caso raro), veja se é algo como "parabéns" ou algo que indique que está quase na flag, aí basta dar:
(gdb) continue
Até a flag aparecer. Ela vai aparecer!
Caso não apareça para você, você também pode rodar:

(gdb) x/s $rdx
(gdb) x/s $rcx
E aí a flag é revelada.

As respostas dos enigmas do quiz:

1. *“O arquivo de usuários não guarda o segredo. O segredo mora no irmão mais restrito (hashes de senha). Pergunta: qual é o caminho absoluto desse arquivo?”*
   Para desvendar esse enigma, precisamos olhar para a estrutura de diretórios padrão do Linux/Unix. Enquanto o arquivo /etc/passwd é o "arquivo de usuários" mencionado (que é legível por quase qualquer pessoa no sistema), o seu "irmão mais restrito" armazena as senhas criptografadas (hashes) e só pode ser lido pelo usuário root. O caminho absoluto do arquivo é /etc/shadow.
   Resposta: /etc/shadow
2. *“Três blocos de 8 bits: 01000011 01010100 01000110. Interprete como ASCII. Pergunta: qual palavra de 3 letras aparece?”*
   Para decodificar essa sequência, convertemos cada bloco binário de 8 bits (1 byte) para seu valor decimal e, em seguida, consultamos a tabela ASCII.
   Passo a passo:
   a. 01000011: $64 + 2 + 1 = 67 = C
   b. 01010100: $64 + 16 + 4 = 84 = T
   c. 01000110: $64 + 4 + 2 = 70 = F

   Resposta: CTF

3. *“Permissões em octal: dono: rwx, grupo: r-x, outros: ---. Pergunta: qual é o modo octal (3 dígitos)?”*
   Para calcular o modo octal, somamos os valores binários atribuídos a cada permissão:
   - r (leitura) = 4
   - w (escrita) = 2
   - x (execução) = 1
   - - (nenhuma) = 0

   O cálculo é:
   - Dono (rwx): 4 + 2 + 1 = 7
   - Grupo (r-x): 4 + 0 + 1 = 5
   - Outros (---): 0 + 0 + 0 = 0

   Resposta: 750
4. *“Redes: quantos endereços TOTAIS existem numa rede IPv4 /27? Pergunta: responda só o número.”*
   Para calcular o número total de endereços em uma sub-rede IPv4, a gente precisa usar uma fórmula baseada nos bits de host restantes. Como um endereço IPv4 possui 32 bits no total, o prefixo /27 indica que 27 bits são destinados à rede, restando 5 bits para os endereços (32 - 27 = 5). O cálculo então é o seguinte:
   2^n, onde n é o número de bits de host
   2^5 = 32

Uma observação interessante é que se a pergunta fosse sobre endereços utilizáveis (para máquinas/hosts), o resultado seria 30, porque subtraímos o endereço de rede (o primeiro) e o broadcast (o último).

Resposta: 32

5. *“IPv4 sem opções: tamanho mínimo do cabeçalho. Pergunta: quantos BYTES?”*
   Para encontrar o tamanho mínimo do cabeçalho IPv4, precisamos observar a estrutura do protocolo. O cabeçalho IPv4 possui um campo chamado IHL (Internet Header Length), que tem 4 bits. Esse campo indica o comprimento do cabeçalho em palavras de 32 bits (4 bytes).
   A estrutura do cabeçalho:
   - Tamanho mínimo: ocorre quando o campo "options" está vazio. Nesse caso, o valor do IHL é 5.
   - Cálculo: 5x4 bytes = 20 bytes

   Resposta: 20
6. *“SHA-256 gera 256 bits. Em hexadecimal, isso vira um tamanho fixo. Pergunta: quantos caracteres tem o digest em hex?”*
   Para converter bits em caracteres hexadecimais, precisamos lembrar que cada dígito hexadecimal (0-9, a-f) representa exatamente 4 bits (2^4 = 16).
   O cálculo: para encontrar o número de caracteres, dividimos o total de bits pelo valor de cada caractere: 256 bits / (4 bits / caractere) = 64 caracteres. Ou seja, o digest (resumo) tem 64 caracteres.

   Resposta: 64
7. *“Existe um sinal POSIX que encerra um processo sem apelação. Não pode ser capturado nem ignorado. Pergunta: nome do sinal (ex: SIGKILL).”*
   Para esse cenário de encerramento forçado, estamos falando do sinal que o kernel utiliza para matar um processo imediatamente, sem dar chance para limpeza de arquivos ou salvamento do estado. Mas é curioso que o "exemplo" dado pelo enigma é justamente a resposta correta. Existem dois sinais do tipo, o SIGTERM (15) e o SIGKILL (9), mas apenas o SIGKILL é não ignorável, não interceptável e tem ação imediata.

   Resposta: SIGKILL
8. *"Uma trilha de HEX foi achada num bilhete: 2f6574632f706173737764. Leia como bytes ASCII. Pergunta: qual é o caminho absoluto resultante?"*
   Para decodificar essa trilha, precisamos converter cada par de dígitos hexadecimais no seu caractere correspondente da tabela ASCII (isso para fazer na mão, para fazer automático é só copiar e colar em um conversor de hex para ASCII).
   O passo a passo da decodificação seria:
   - 2f representando "/"
   - 65 representando "e"
   - 74 representando "t"
   - 63 representando "c"
   - 2f representando "/"
   - 70 representando "p"
   - 61 representando "a"
   - 73 representando "s"
   - 73 representando "s"

- 77 representando "w"
- 64 representando "d"

Ou seja, o caminho absoluto resultante é: /etc/passwd.

Resposta: /etc/passwd

9. *“DNS: qual registro (uma única letra) aponta nomes para IPv4? Pergunta: qual é a letra?”*

   No sistema de nomes de domínio (DNS), diferentes tipos de registros são usados para mapear informações específicas. Quando queremos traduzir um nome de domínio (como google.com) para um endereço IPv4 de 32 bits, utilizamos um registro específico, o registro "A". O nome vem de "Address" (Endereço). No IPv6, que tem 128 bits, ou seja, quatro vezes o tamanho do IPv4, o registro é o quad-A, ou seja, o AAAA.

   Resposta: A

10. *“No /proc existe um arquivo que lista estatísticas das interfaces (RX/TX). Pergunta: qual é o caminho completo?”*

    Para encontrar as estatísticas de tráfego de rede (recebimento e transmissão) diretamente no sistema de arquivos virtual (o /proc), precisamos acesar o arquivo que mapeia o status dos dispositivos de rede. O arquivo é o /proc/net/dev, inclusive, é lá que se guarda uma tabela detalhada com colunas para cada interface (como eth0 ou lo). Os comandos como ifconfig ou ip -s link retiram seus dados exatamente desse arquivo.

    Resposta: /proc/net/dev

11. *“Um byte x: x = 10110110 (binário). Faça: x >> 3. Pergunta: resultado em binário com 8 bits (com zeros à esquerda)?”*

    Para resolver essa operação, realizamos um deslocamento de bits à direita (right shift). O operador ">> 3" move todos os bits três posições para a direita. Os três bits menos significativos (à direita) são descartados, e preenchemos as três novas posições à esquerda com zeros (considerando um deslocamento lógico).

    O processo:

    1. Original: 10110 110
    2. Deslocando 1: 01011011
    3. Deslocando 2: 00101101
    4. Deslocando 3: 00010110

    Inclusive, uma curiosidade matemática é que deslocar 3 bits para a direita é equivalente a dividir o número original por 2^3 (ou seja, dividir por 8) e descartar o resto.

    Resposta: 00010110

12. *“Um comando foi escrito em BINÁRIO (ASCII, 8 bits por letra): 01100011 01101000 01101101 01101111 01100100. Pergunta: decodifique e responda o comando (apenas a palavra).”*

    Para decodificar esse comando, convertemos cada grupo de 8 bits para seu valor decimal e, em seguida, para o caractere correspondente na tabela ASCII.

    Passo a passo:

    1. 01100011 = 64 + 32 + 2 + 1 = 99 = c
    2. 01101000 → 64 + 32 + 8 = 104 = h
    3. 01101101 → 64 + 32 + 8 + 4 + 1 = 109 = m
    4. 01101111 → 64 + 32 + 8 + 4 + 2 + 1 = 111 = o
    5. 01100100 → 64 + 32 + 4 = 100 = d

    Resposta: chmod

13. *“Quatro bytes apareceram num dump (hex): 78 56 34 12. Interpretados como inteiro 32-bit little-endian, viram outro hex. Pergunta: qual é o valor final em hex (sem 0x, minúsculo)?”*
    Para resolver este enigma, precisamos inverter a ordem dos bytes conforme a convenção Little-Endian. Em sistemas Little-Endian, o byte menos significativo (LSB) é armazenado no endereço de memória mais baixo. Quando lemos esses bytes como um único número inteiro de 32 bits, devemos ler a sequência de trás para frente (do último byte para o primeiro).
    Processo de inversão:
    1. Dump original (bytes na memória): 78 (pos 0), 56 (pos 1), 34 (pos 2), 12 (pos 3).
    2. Reordenação Little-Endian: O último byte lido torna-se o mais significativo.
    3. Resultado: 12 34 56 78
    Resposta: 12345678
14. *“Regex: qual símbolo ancora a busca no INÍCIO da linha? Pergunta: responda com um único caractere.”*
    Para ancorar uma expressão regular no início de uma linha, utilizamos o símbolo de circunflexo (^). Exemplo de uso: "^Texto" corresponde a qualquer linha que comece com a palavra "Texto". E "Texto$" coresponde a qualquer linha que termine com a palavra "Texto" (usando o âncora de fim de linha).
    Resposta: ^
15. *"Sou o único número que veste a capa do superusuário no Linux. Pergunta: qual é o UID (apenas o número)?"*
    Essa questão trata da identidade numérica do administrador do sistema. No Linux, os nomes de usuário são apenas "rótulos" para humanos, o sistema mesmo vai identificar quem você é através do UID (User ID). O UID reservado exclusivamente para o usuário root é 0. O kernel do Linux trata o processo com UID 0 de forma especial, permitindo que ele ignore quase todas as verificações de permissão de arquivos e privilégios de rede. E mesmo que você renomeie o usuário "root" para outro nome, se o UID dele permanecer 0, ele continuará sendo o superusuário.
    Resposta: 0

Flag final: dashboard