OverTheWire

Writeups dos jogos do overthewire: http://overthewire.org/wargames/

- Leviathan
- Narnia
- Natas
- Krypton

Leviathan

1->2

strace e ltrace (identificam as system calls e shared library calls)

2->3

rodar dentro de /etc/leviathan_pass com um arquivo com espaço no nome terminando em leviathan3 (e.g. '/tmp/dir/foo leviathan3')

5->6

criar um link simbólico do arquivo que vc quer ler pro arquivo que o programa lê

6->7

usar 'objdump -d' para ver o valor contra o qual o programa compara

Narnia

Informações gerais

"narnia.labs.overthewire.org through SSH on port 2226"

Todos os níveis têm seus programas e códigos em /narnia.

narnia0

```
$ ( echo -e "0123456789012345678 9\xef\xbe\xad\xde" ; cat ; ) | /narnia/narnia0
```

narnia1

colocar um shellcode na variavel de ambiente EGG para ler o arquivo /etc/narnia_pass/narnia2 (vide shellcode.asm em anexo)

e.g. para injetar 0x01 0x02:

```
        $ export EGG="$(echo -ne '\x01\x02')"
```

Conferimos o shellcode com

```
        $ echo -n "$EGG" | hedxump -C
```

narnia2

stack smashing: escrever um número N de bytes, seguido do endereço B do buffer, para sobrescrever o endereço de retorno da função na stack frame, igual ao jumpy2 do limbo. Colocamos um shellcode nesses N bytes para ler /etc/narnia_pass/narnia3.

O valor de N é a diferença entre o endereço B do buffer e o endereço R da pilha no qual está guardado o endereço de retorno de main().

Para achar R: Rodando no gdb podemos ver centenas de bytes a partir do endereço que estiver

em esp imediatamente antes da chamada de strcpy(). Com isso acharíamos R, mas não sabemos de qual endereço main() foi chamada. Sabemos que foi da função __libc_start_main(), então basta ver os endereços do começo e do final dessa função e procurar na pilha um valor nesse intervalo, e cuja instrução imediatamente anterior seja call.

Para checar que o valor de N está certo, rodamos de novo no gdb, com N bytes seguidos de 4 bytes fixos, e vemos se o programa vai parar lá. Fazendo isso, vemos que N = 140.

Finalmente, precisamos achar B, que varia de acordo com o tamanho da string que passamos como argumento. Agora, já sabemos que devemos escrever uma string com N+4 bytes (porque queremos N bytes seguidos do endereço B, e como o programa está em 32 bits, B tem 4 bytes). Então para achar B, basta rodar narnia2 no ltrace, com uma string de N+4 bytes como argumento; B será o endereço mostrado como destino de strcpy(). Depois é só escrever o shellcode para ler o arquivo que da senha narnia3, preencher os bytes que faltam com NOP para ter N bytes, e colocar o endereço B em seguida. Passamos o shellcode da mesma maneira que o anterior, com

```
$(echo -ne '\x<byte1>\x<byte2>...')
```

Vide shellcode2.asm em anexo.

narnia3

outro stack smashing, mas dessa vez devemos sobrescrever a string ofile (vide /narnia/narnia3.c) para que a senha seja escrita em um arquivo que podemos ler. Ao mesmo tempo, temos que passar como ifile o caminho do arquivo que tem a próxima senha: /etc/narnia_pass/narnia4.

Para sobrescrever o ofile, basta colocar várias '/' depois do caminho do ifile, até dar 32 caracteres; tudo depois disso será visto como o ofile. Por exemplo:

```
$ /narnia/narnia3 "/etc/narnia_pass/narnia4/////////tmp/foo"
                   ^1                            32^
```

O problema: a string do ofile também vai fazer parte do ifile. Nesse caso o ofile será /tmp/foo e o ifile será a string inteira: /etc/narnia_pass/narnia3/////////tmp/foo, porque a string só acaba no terminador nulo de ofile. Como estamos passando essa string como um parâmetro no bash, não dá para colocar um '\0' antes de começar o ofile, porque esses parâmetros também são armazenados como strings, e um '\0' no meio da string faria com que ela acabasse antes do que

devia.

Primeiramente, o comando acima é equivalente a

```
$ /narnia/narnia3 /////////etc/narnia_pass/narnia4/tmp/foo
```

Contanto que o número de barras seja igual. O que fizemos, então, foi usar caminhos relativos a nosso favor: em vez de tentar escrever em /tmp/foo, podemos criar um arquivo /tmp/narnia4 e rodar, em /tmp, esse comando:

```
$ /narnia/narnia3 "..///////////////etc/narnia_pass/narnia4"
                   ^1                            32^
```

De modo que o ofile seria narnia4 (ou seja, /tmp/narnia4, porque estamos em /tmp) e o ifile seria /tmp/..///////////////etc/narnia_pass/narnia4 (novamente porque estamos em /tmp). Daí vem um segundo problema: o arquivo /tmp/narnia4 já existe, e é um link simbólico para /etc/narnia_pass/narnia4; não podemos escrever nele. A maneira mais fácil de resolver isso é colocando mais 2 barras...

```
$ /narnia/narnia3 "..////////////////etc/narnia_pass/narnia4"
                   ^1                            32^
```

...para que o ofile passe a ser s/narnia4. Então basta criar a pasta s/ e dentro dela o arquivo narnia4, rodar o comando, e ler s/narnia4.

Natas

11->12

level exige que se faça upload de um jpeg, mas não checa se é mesmo um jpeg; dá pra fazer upload de um script php

- O problema é, esse script é sempre renomeado para um string com .jpg no fim; na hora de clicar no link o browser não executa o script
- MAS, a string com o nome final do arquivo está na verdade em um <input> com atributo hidden, dá pra editar e colocar extensão .php
- Com isso dá pra printar a senha em /etc/natas_webpass/natas13

12->13

- Parecido com o anterior, mas ele realmente checa se a imagem é um jpeg
- Ainda dá pra upar um script editando os primeiros bytes do script pra ser igual ao magic number do jpeg
- Pra adicionar bytes num arquivo, o melhor jeito é usar o bvi (binary vi)

13->14

envolve SQL injection; é preciso injetar código SQL (parecido com XSS) no campo username para garantir que a query vai retornar pelo menos uma linha (mysql_num_rows() > 0). O melhor jeito de fazer isso é garantir que a condição da query é sempre true. Para isso, podemos fechar as aspas, colocar OR true e # (comentário).

14->15

SQL injection, LIKE operator, LIKE BINARY (case-sensitive); vide script15

15->16

brutar cada caractere da senha, com grep e regex; vide script16

16->17

parecido com o 16, mas com sql (RLIKE BINARY). Usamos sleep() para saber se o caractere da senha está certo. Vide script17.py

17->18

como o ID máximo de usuário é 640, fizemos um javascript para atualizar o valor do cookie PHPSESSID, para tentar todos os IDs possíveis até chegar no ID do admin. (138?)

Krypton

Krypton0

Precisamos decodificar a mensagem `S1JZUFRPTklTR1JFQVQ=` para o próximo nível.

É dito que ela está em **base64**. Para resolver isso, é só usar o comando `base64`.

```
echo "S1JZUFRPTklTR1JFQVQ=" | base64 -d
```

Assim, obtemos a senha de Krypton1.

Resposta: `KRYPTONISGREAT`.

Krypton1

É preciso decodificar a mensagem `YRIRY GJB CNFFJBEQ EBGGRA`, encriptada por **rotação simples**.

Rotação simples pode significar alguma versão da Cifra de César, provavelmente ROT13. Para isso, podemos usar sites como dcode ou criar um programa em python que automatize isso:

```
def rot(char, shift):
    return chr((ord(char) - ord('A') + shift)%26 + ord('A'))

def decode(ciphertext, shift):
    msg = ''
    for c in ciphertext:
        msg += rot(c, shift) if c.isalpha() else c
    return msg

ciphertext = 'YRIRY GJB CNFFJBEQ EBGGRA'
shift = 13
flag = decode(ciphertext, shift)
```

```
print(flag)
```

A resposta está na cifra decodificada por ROT13: `LEVEL TWO PASSWORD ROTTEN`.

Resposta: `ROTTEN`.

Krypton2

Temos no arquivo `krypton3` a mensagem `OMQEMDUEQMEK` que está criptografada em **Cifra de César**.

Há duas formas de resolver esse nível.

Podemos criar uma pasta em `/tmp/` e linkar a keyfile nele para usarmos o executável `encrypt`. Dessa forma, podemos encriptar um arquivo com `a` para descobrir a rotação usada.

Com isso, o arquivo encriptado terá a letra `m`, indicando que foi usado ROT12 para encriptação e deverá ser usado ROT14 para decriptação, obtendo a *flag*.

Outro modo, muito mais rápido, é usar o mesmo método do nível anterior, fazendo um *brute-force* das 26 rotações possíveis de uma Cifra de César.

```
def brute_force(ciphertext):
    for shift in range(26):
        print(decode(ciphertext, shift))

brute_force('OMQEMDUEQMEK')
```

A que tiver a frase mais legível é a *flag*.

Resposta: `CAESARISEASY`.

Krypton3

Para esse nível, temos no arquivo `krypton4` a mensagem `KSVVW BGSJD SVSIS VXBMN YQUUK BNWCU ANMJS`

, cifrada por alguma **cifra de substituição simples**. Nesse momento, usar *brute-force* não é mais uma opção.

Para resolver isso, podemos pegar o texto em `found1` e aplicar uma análise de frequência em suas letras.

Uma ferramenta online extremamente eficiente está localizada em guabala.de. Utilizando sua análise de frequência, obtemos o mapeamento:

```
abcdefghijklmnopqrstuvwxyz     This clear text ...
qazwsxedcrfvtgbyhnujmikolp     ... maps to this cipher text
```

Podemos, assim, decriptar a mensagem com o site dcode utilizando o alfabeto de susbtituição acima. O resultado é o texto `WELLD ONETH ELEVE LFOUR PASSW ORDIS BRUTE`.

Resposta: `BRUTE`.

Krypton4

Nesse nível, há a mensagem `HCIKV RJOX` em `krypton5`, codificada pela **Cifra de Vigenère**.

Primeiro, precisamos descobrir a *key* da cifra. Para isso usaremos o texto em `found1` e a ferramenta em dcode. Assim, encontramos a *key* `FREKEY`.

Dessa forma, utilizando novamente o dcode com a *key*, obtemos a *flag*.

Resposta: `CLEARTEXT`.

Krypton5

Para esse nível, também temos uma mensagem codificada pela **Cifra de Vigenère**, porém sem termos o conhecimento do tamanho da chave.

Para descobrir a *key* podemos usar o texto em `found1` e a ferramenta em dcode. Usando aanálise estatística, obtemos uma resposta parcial: `KEYLEBGTH`.

Essa resposta é próxima de um termo legível: `KEYLENGTH`. Testando essa *key* em `found1` usando o dcode, vemos que ela decifra o texto perfeitamente.

Assim, decriptando o texto `BELOS Z` usando a *key* `KEYLENGTH` obtemos a flag.

Resposta: `RANDOM`.

Krypton6

Nesse último nível, a mensagem em `krypton7` está codificada pelo método de One-Time Pad, usando o programa `encrypt6`.

Como temos o programa, podemos usar o método do `Chosen-plaintext attack` para descobrir seu funcionamento.

Primeiro, criamos uma pasta em `/tmp/` para os arquivos desse nível

```
mkdir /tmp/files/
```

Depois, vamos criar um arquivo apenas com "A"s, para ser encriptado.

```
echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" > /tmp/files/a
```

Assim, encriptando ele por

```
./encrypt6 /tmp/files/a /tmp/files/out
```

obtemos a sequência `EICTDGYIYZKTHNSIRFXYCPFUEOCKRNEICTDGYIYZKTHNS`. Ao realizar o mesmo comando várias vezes, a resposta continua a mesma. Logo, a *key* não é aleatória.

Além disso, vemos que ela se repete produzindo a sequência `EICTDGYIYZKTHNSIRFXYCPFUEOCKRN` várias vezes.

Testando com um arquivo com apenas "B"s,

```
echo "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB" > /tmp/files/b
./encrypt6 /tmp/files/b /tmp/files/out2
```

Obtemos a sequência `FJDUEHZJZALUIOTJSGYZDQGVFPDLSOFJDUEHZJZALUIOT`, que é exatamente a anterior com um shift a mais no alfabeto em cada caractere.

Assim, temos que o One-Time Pad não é aleatório, usa a mesma seed, cuja *key* repete ao longo da mensagem, e se comporta com shifts no alfabeto, como na cifra de Vigenère.

Por fim, um pequeno script em Python consegue recuperar a flag:

```
input = 'PNUKLYLWRQKGKBE'
key = 'EICTDGYIYZKTHNS'
base = ord('A')
flag = ''
for c_in, shift in zip(input, key):
    flag += chr( (ord(c_in) - ord(shift) - 2*base)%26 + base)

print(flag)
```

Resposta: `LFSRISNOTRANDOM`.

Outros Write-ups e arquivos

- Nada ainda