CNS Homework 3

B11902083 徐粲桓

1. Welcome To Fuzzing!

- Reference: All by myself.

(a)

Grey-box fuzzing (AFL fuzzer as example)

The gray-box fuzzing process consists of the following steps:

1. The test program is processed by instrument program to trace basic block (a code sequence without branches) transition of an input during compilation by `afl-gcc`.
2. Test inputs is selected from the input seed corpus to seed queue
3. A seed is selected, trimmed and mutated to become a test case
4. Test cases are ran and its coverage/crashes/hanging... are recorded
5. If the mutated inputs gave an updated coverage, it's fed back to the seed queue.
6. Repeat from step 3.

Mutation-based vs. Generation-based fuzzers

Mutation-based fuzzers aim to increase chances to obtain valid inputs, by starting with a set of existing valid inputs as seeds and creating new inputs by applying "mutations" (i.e. small changes) to the seeds. AFL is an example of mutation-based fuzzers.

Generation-based fuzzers aim to test programs of highly structured inputs (e.g. PDF, XML) by providing pre-defined grammar rules (e.g. structure, relationships of data) to build a model that generates structurally correct test cases and gives better code coverage. Peach Fuzzer is an example.

(b)

I collected the crashes with command `afl-fuzz -i input_seeds -o out -- ./app -i @@`, therefore, the POC here is relative from the path `out/default`. The files will be included in `b11902083/code/` for reference.

Vulnerability #1

The POC `crashes/id\:000007\,sig\:11\,src\:000125\,time\:3633902\,execs\:2725144\,op\:havoc\,rep\:9` triggers a **null pointer reference**. I used AddressSanitizer to analyze this issue. The setup is:

- Compile with `gcc app.c -fsanitize=address,undefined -o app_clean`
- Run with `./app_clean -i <POC>`

The output of ASan is as follows:

```
app.c:96:10: runtime error: load of null pointer of type 'int'
AddressSanitizer:DEADLYSIGNAL
=================================================================
==91==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x6416909af9c9 bp 0x7ffd9bb06b30 sp 0x7ffd9bb06b00 T0)
==91==The signal is caused by a READ memory access.
==91==Hint: address points to the zero page.
    #0 0x6416909af9c9 in dHGWZ3BU0MMSmMqXhNDLey7N (/src/app_clean+0x389c9)
    #1 0x6416909bef89 in tK3toloRqLDW1CsaX8dixbrL (/src/app_clean+0x47f89)
    #2 0x6416909ca97d in kNE1mBq3WSvg2CmrQiqDGSJG (/src/app_clean+0x5397d)
    #3 0x6416909cae45 in main (/src/app_clean+0x53e45)
    #4 0x75e796c27d8f  (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f)
    #5 0x75e796c27e3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f)
    #6 0x6416909af524 in _start (/src/app_clean+0x38524)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/src/app_clean+0x389c9) in dHGWZ3BU0MMSmMqXhNDLey7N
```

It appears that this PoC triggered a null pointer reference in the function `dHGWZ3BU0MMSmMqXhNDLey7N()`, which is essentially a swap of two integers. The swap is called in `tK3toloRqLDW1CsaX8dixbrL()`, and the local variable `int* cv8a78xzjs81ka8xka8fs` was initialized `NULL` before passed into swap, thus the null pointer reference.

```c
// in tK3toloRqLDW1CsaX8dixbrL()
int *cv8a78xzjs81ka8xka8fs = NULL;
dHGWZ3BU0MMSmMqXhNDLey7N(jaisdfjiojasidofnains, cv8a78xzjs81ka8xka8fs);
```

Vulnerability #2

The POC `crashes/id\:000001\,sig\:11\,src\:000001\,time\:22630\,execs\:19704\,op\:havoc\,rep\:5` triggers a **global-buffer-overflow**. I also used ASan here, same steps as #1.

The output of ASan is:

```
app.c:367:86: runtime error: index 1000 out of bounds for type 'char [1000]'
app.c:367:33: runtime error: index 1000 out of bounds for type 'char [1000]'
app.c:367:86: runtime error: load of address 0x5e70d5a2db08 with insufficient space for an object of type 'char'
0x5e70d5a2db08:       pointer points here
 1a 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00
              ^

=================================================================
==35==ERROR: AddressSanitizer: global-buffer-overflow on address 0x5e70d5a2db08 at pc 0x5e70d59bedea bp 0x7fff7f3a18d0 sp 0x7fff7f3a18c0
READ of size 1 at 0x5e70d5a2db08 thread T0
    #0 0x5e70d59bede9 in alpKX7wAPFzJbXzjZTOQi4ym /src/app.c:367
    #1 0x5e70d59d09e7 in kNE1mBq3WSvg2CmrQiqDGSJG /src/app.c:1910
    #2 0x5e70d59d0eaf in main /src/app.c:1948
    #3 0x7ec3d9b5bd8f  (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f)
    #4 0x7ec3d9b5be3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f)
    #5 0x5e70d59b5524 in _start (/src/app_clean+0x38524)

0x5e70d5a2db08 is located 0 bytes to the right of global variable 'rm7QkP0YhYIxpk8hjeRtvTRA' defined in 'app.c:25:6' (0x5e70d5a2d720) of size 1000
0x5e70d5a2db08 is located 56 bytes to the left of global variable 'Rz45kGxh0fWbr2uZQ22PNpkF' defined in 'app.c:26:6' (0x5e70d5a2db40) of size 1000
SUMMARY: AddressSanitizer: global-buffer-overflow /src/app.c:367 in alpKX7wAPFzJbXzjZTOQi4ym
```

By analyzing the source according to ASan's output, the error happens in this for loop at `alpKX7wAPFzJbXzjZTOQi4ym()`:

```
// in alpKX7wAPFzJbXzjZTOQi4ym()
for(xUJ3fFc1NxYNEjt6X4W0pWvQ = 0; xUJ3fFc1NxYNEjt6X4W0pWvQ < jUntIhmLCWXoIzkaPaqFjth8;
xUJ3fFc1NxYNEjt6X4W0pWvQ++){
        IQV6hMMwJWidC7NakjKv929e[xUJ3fFc1NxYNEjt6X4W0pWvQ] = rm7QkP0YhYIxpk8hjeRtvTRA[xUJ3fFc1NxYNEjt6X4W0pWvQ];
    }
```

where `IQV6hMMwJWidC7NakjKv929e` is an array of size 1000. Apparently it is because the variable `jUntIhmLCWXoIzkaPaqFjth8 >= 1000` causing the index to be out of bound. To confirm, I then used GDB in following steps:

- Compile with debugging info: `gcc -g app.c -o app_gdb`
- Run with `gdb app_gdb`
    - `set args -i <POC>`
    - `break alpKX7wAPFzJbXzjZTOQi4ym`
    - `run`
    - `print jUntIhmLCWXoIzkaPaqFjth8` -> `$1 = 2319`

Vulnerability #3

The POC `hangs/id\:000000\,src\:000001\,time\:1167\,execs\:297\,op\:quick\,pos\:18` triggers **infinite-loop (hangs)**. Here I used GDB in the following steps:

- `gcc -g app.c -o app_gdb`
- `gdb app_gdb`
    - `run`
    - `SIGINT` (ctrl-c) when hang, shows the current line 713.
    - `list`

The output of `list` was the following:

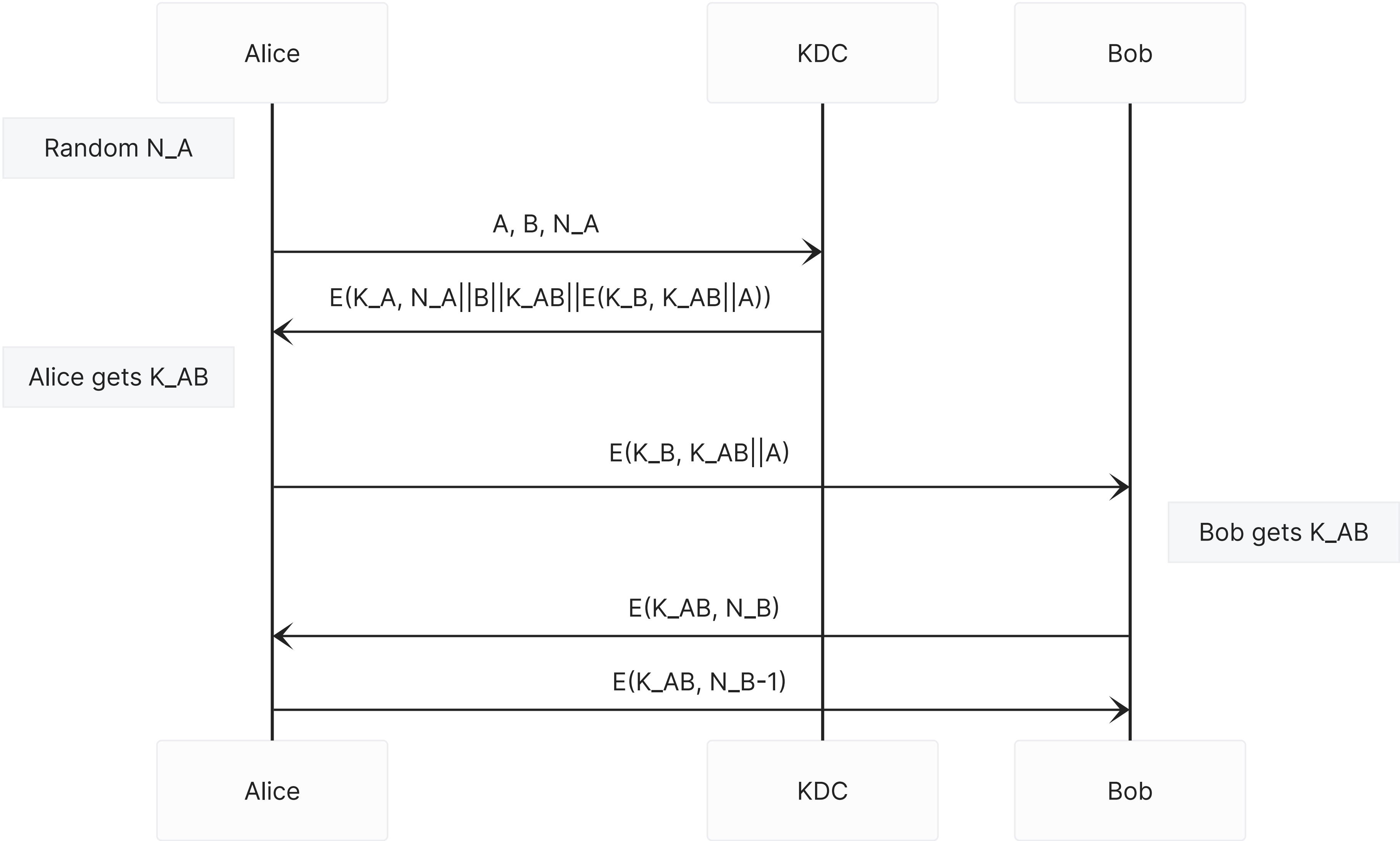
```
(gdb) list
708         a0sjDf1nQGsfNcjrnoqfeYkEE = a0sjDf1nQGsfNcjrnoqfeYkEE * a18XDi2UNOZGX5ZfWQUMcng4f;
709         vBWO7wt69PQ6A6YOLE7RLksY = vBWO7wt69PQ6A6YOLE7RLksY + a98gzkiF94ph1X4DOpiYjLpMu;
710         LfEL1RFxW94NUwWGjxL00Zos = LfEL1RFxW94NUwWGjxL00Zos + xaxT3POIWSGDUEzktAvHpQnT;
711         dHGWZ3BU0MMSmMqXhNDLey7N(&a18XDi2UNOZGX5ZfWQUMcng4f, &a98gzkiF94ph1X4DOpiYjLpMu);
712         LfEL1RFxW94NUwWGjxL00Zos = 1;
713         while(LfEL1RFxW94NUwWGjxL00Zos){
714             if( LfEL1RFxW94NUwWGjxL00Zos < 100 ) LfEL1RFxW94NUwWGjxL00Zos++;
715         }
```

This while loop increases `LfEL1RFxW94NUwWGjxL00Zos` every iteration until it reaches `LfEL1RFxW94NUwWGjxL00Zos == 100`, however, does not break out the loop, causing infinite loop as `LfEL1RFxW94NUwWGjxL00Zos > 0`.

2. Needham-Schroeder Protocol

- Reference: All by myself.

(a)



For implementation detail, please refer to `code/code2.py`.
The flag: `CNS{N33DH4M_5CHR03D3R_PR070C0L_15_4W350M3_8e7c1985126d2142b87cdbc8ccca86aa}`

(b)

Explanation

Offline password cracking is the process of cracking passwords from hashed representations of passwords (here, it is SHA-256 hashed) without interacting directly with the service where the passwords are used.

Flag

We'll use John the Ripper to crack the password. By checking `john --list=formats` we know we can use `raw-sha256` as hash type. The full command is `john --wordlist=common_roots.txt --format=raw-sha256 hashfile` where the hash is stored in single line in `hashfile`, and `common_roots.txt` is downloaded from [this repo](). The password is `m45t3rm1nd`.

The flag: `CNS{D0N_7RY_7H15_47_H0M3_e340c50d4e72213257bab1a0deb4a2bd}`

(c)

Flag

For implementation detail, please refer to `code/code2.py`.
The flag: `CNS{R3PL4Y_4774CK_15_3V3RYWH3R3?!_d469fa0bd241f15122bc3aa38e8ed7ee}`

Explanation & Fix

The protocol is vulnerable to replay attack. If the attacker has recover a past session key K_{AB} and the forwarded message $E(K_B, K_{AB}||A)$ from Alice to Bob, then, the attacker can finish the rest of communication (starting from the forwarded message) and let Bob think they're talking to Alice with session key K_{AB}.

The main reason of this vulnerability is due to the design of this protocol, as the order of communication caused Bob unable to check the freshness of key with its own nonce (Bob has to accept the key as long as the forwarded message can be decrypted with K_B.)

A fix to this is that KDC server and the clients (Alice & Bob) synchronizes a timestamp t between them, and include the timestamp in the forwarded message $E(K_B, K_{AB}||A||t)$. This allows Bob to check the freshness of K_{AB} and abort if he finds an outdated key.

3. Accumulator

- Reference
  - How to find the embedding degree of an elliptic curve?
  - Consulted sage usage from B11902054 周得壹

For convenience, we define $A = \Pi_{t \in S} t, d(S) = g^A$ in (a), (b).

(a)

Given p, q one can compute the modulus of RSA $\phi(n) = (p-1)(q-1)$.
Our goal is to fake a membership proof for some prime $s \notin S$ i.e. $1 = \gcd(s, \phi(n))$
$\Rightarrow$ By Extended Euclidean algorithm, we can find $\exists a, b$ s.t. $as + bn = 1$
$\Rightarrow as \equiv 1 \bmod \phi(n), a = s^{-1}$
$\Rightarrow$ Let proof

$$\pi = d(S)^a = g^{\Pi_{t \in S \setminus \{s\}} t \cdot (as) \bmod \phi(n)} = g^{\Pi_{t \in S \setminus \{s\}} t} = g^A$$

Verify that $\pi^s = d$, indeed.

(b)

For $w \in S$, similar to what we done in (a), by Extended Euclidean algorithm:
$\Rightarrow \exists a$ s.t. $aw \equiv 1 \bmod \phi(n), a = w^{-1}$
$\Rightarrow$ Let proof $\pi = (\pi_1, \pi_2)$, where

$$\pi_1 = g^a, \pi_2 = 0$$

Verify that $(g^a)^w \cdot d^0 = g^{aw} \cdot 1 = g^1 = g$, indeed.

For convenience, we define $A = \Pi_{t \in S}(c - t), d(S) = g_1^A$ in (c-f).

(c)

Given bi-linear function f, digest $d(S)$, generator g_2, g_2^c, one can verify membership proof π for member s:

$$f(\pi, g_2^{(c-s)}) \stackrel{?}{=} f(d(s), g_2)$$

Suppose for the member $s \in S$, the membership proof $\pi = g^{\Pi_{t \in S \setminus \{s\}}(c-t)}$ is given, then:

$$f(\pi, g_2^{(c-s)}) = f(g_1, g_2)^{\Pi_{t \in S \setminus \{s\}}(c-t) \cdot (c-s)} = f(g_1, g_2)^A = f(g_1^A, g_2) = f(d(S), g_2)$$

passes the verification.

(d)

For a fake member $w \notin S$, knowing c allowing us to know $c - w$ and thus we can find $(g_1^{(c-w)})^{-1}$, and raise it to A-th power. Then, let the fake membership proof π be:

$$\pi = g_1^{A(c-w)^{-1}}$$

The verification:

$$f(\pi, g_2^{(c-s)}) = f(g_1, g_2)^{A(c-s)^{-1}(c-s)} = f(g_1, g_2)^A = f(g_1^A, g_2) = f(d(S), g_2)$$

still stands, thus creating a fake membership proof.

(e)

Given bi-linear function f, digest $d(S)$, generator g_1, g_2, one can verify non-membership proof $\pi = (\pi_1, \pi_2)$ for a non-member u:

$$f(\pi_1, g_2^{(c-u)}) \cdot f(g_1, g_2)^{\pi_2} \stackrel{?}{=} f(d(s), g_2)$$

or alternatively, by shifting the term to right:

$$f(\pi_1, g_2^{(c-u)}) \stackrel{?}{=} f(g_1^{A-b}, g_2)$$

Note that $A = p(c)$ by our definition.
Suppose for a non-member $u \notin S$, the non-membership proof $\pi = (g_1^{q(c)}, b)$ is given, then:

$$f(\pi_1, g_2^{(c-u)}) = f(g_1, g_2)^{q(c)(c-u)} = f(g_1, g_2)^{p(c)-b} = f(g_1^{A-b}, g_2)$$

passes the verification.

(f)

For $w \in S$, we know that $b = 0$ as $p(x)/(x - w)$ don't give remainder anymore. Furthermore, knowing c allow us to compute $q(c) = d(S)/(c - w)$. Then, let the non-membership proof $\pi = (\pi_1, \pi_2)$ be:

$$\pi_1 = g_1^{q(c)}, \pi_2 = 0$$

The verification:

$$f(\pi_1, g_2^{(c-w)}) = f(g_1, g_2)^{q(c)(c-w)} = f(g_1, g_2)^{p(c)} = f(g_1^{A-0}, g_2)$$

still stands, thus creating a fake non-membership proof.

(g)

Please refer to `code/code3.py`.

Generate Membership Proof

After building the curve, simply follow the faking of membership proof in (d).
The flag: `CNS{bI1iN34r_4cCumU14tOr5_4rE_FUn}`

Generate Non-Membership Proof

Similar to membership proof, but this time follows the faking of non-membership proof in (f)
The flag: `CNS{311ipTIc_CUrV3S_Ar3_4lS0_FuN}`

Verify Membership Proof

g_2 is in form of p-ary polynomials up to 12 degree, meaning the G_2 here is over $\mathbb{F}_{p^{12}}$, we will build both E_1, E_2 on $\mathbb{F}_p$ and $\mathbb{F}_{p^{12}}$ respectively. In sage we can calculate the Weil's pairing $f(g_1, g_2)$ with `g1.weil_paring(g2, E1.order())` for any $g_1 \in G_1, g_2 \in G_2$.

After constructing all members for verification, simply check if the condition in (c) is true/false.

The flag: `CNS{Y0u_r34Lly_kN0w_mY_m3MbEr5}`

Verify Non-Membership Proof

Similar to membership proof verification, but (e) this time.
The flag: `CNS{Y0u_c4N_b3_mY_s3cUr1Ty_gU4rD}`

(h)

By consider the Weil's paring, we can map points on the elliptic curve group E to a multiplicative group $\mathbb{F}_{p^k}$ where k is the embedding degree. Since normally k is large and solving discrete log problem in $\mathbb{F}_{p^k}$ is not easier, but with small k, to find n s.t. $nP = Q, P, Q \in E$, we can follow the steps:

- Compute Weil's pairing $f(P, Q) = u \in \mathbb{F}_{p^k}$
- $f(nP, Q) = v = nf(P, Q) = nu \in \mathbb{F}_{p^k}$
- It suffices to solve n s.t. $v = nu$ in $\mathbb{F}_{p^k}$, which is easier to solve than that of elliptic curve.

(i)

Please refer to `code/code3.py` for embedding degree calculation, which is at option 5.

- `secp256k1`: $k = 2$, is vulnerable
- `Curve25519`: $k = 19$, is not vulnerable.

4. DDoS

- Reference: Discussed with B11902050 范綱佑 and this article.

(1)

The traffic shows that the first packet sent is **the packet No. 55863 at time 24.945277**, which can be found by observing the I/O graphs near the first burst of traffic (making the interval smaller gives more accurate timestamp) The victim IP is `192.168.232.95`.

(2)

The attacker exploits the UDP protocol and the size of attack packet is 482.
As for the application layer protocol, it's probably NTP as we can observe the attacking UDP packets has source port of 123 (NTP) and destination port of 443 (HTTPS). We cannot determine whether it is using monlist (although very likely) since the packets are truncated during capture.

(3)

Victim is a (web) server, which can be observed from several client connection attempts before the attack packet arrives. The intention is to paralyze the web server by overwhelming port 443.

(4)

By analyzing IPv4 Statistics and check what destination address received most packet, there're total 3 victims: `192.168.232.10`, `192.168.232.80`, `192.168.232.95`.
To find out which amplifier sent most packet, we can use `Statistics > Conversation` and check the `UDP` tab, sort the conversation by descending order and for each victim, identify the top 3 address that sent most packet.

`192.168.232.10`

- Number of packet sent to victim: `26870`
- Three major amplifiers
  - `34.93.220.190` with 500 packets
  - `5.104.141.250` with 492 packets
  - `124.120.108.157` with 414 packets

`192.168.232.80`

- Number of packet sent to victim: `28320`
- Three major amplifiers
  - `128.111.19.188` with 538 packets
  - `124.120.108.157` with 500 packets
  - `129.236.255.89` with 500 packets

`192.168.232.95`

- Number of packet sent to victim: `23327`
- Three major amplifiers
  - `34.93.220.190` with 500 packets
  - `128.111.19.188` with 500 packets
  - `212.27.110.13` with 403 packets

(5)

Here I chose `82.65.72.200` as the amplifier (the port's availability can be check with `nmap -sU <ip> -p 123` in advance) and send the `monlist` query with `ntpdc -n -c monlist 82.65.72.200`. For packet capturing, I used `tcpdump` and limited captured traffic to my wired interface `enp5s0`, the full command is `sudo tcpdump -i enp5s0 udp port 123 and host 82.65.72.200 -w ddos.pcapng`.

Analyzing the file `ddos.pcapng`, I found that my request was of size `234` and the server replied with 100 packets of size `482`. The amplification factor r can thus be computed by the ratio of response size to request size:

$$r = \frac{48200}{234} \approx 205.982$$

(6)

Amplifier

For an amplifier's server operator, they can:

- Disable or limit unauthorized queries of certain command like `monlist`
- Rate limiting the number of requests for NTP server over UDP from a single IP.
- Flow monitoring to identify abnormal requests (e.g. A large flow of `monlist` requests)
- Firewall to block the source IP identified from flow monitoring

Victim's network

For the network administrator of victim's network, they can:

- Ingress filtering to detect IP spoofing as UDP allows easily spoofing source IP
- Rate limiting, but this time limit the response from NTP servers
- Use BGP redirecting traffic to a blackhole

5. Private Information Retrieval

- Reference: All by myself

I only done (a), (b), please refer to `code/code5.py`.

(a)

Simply follow the instruction as client side, for details please refer to `code/code5.py`.
The flag: `CNS{1nf0rmat10n_th30r3t1cally_s3cur3}`

(b)

If we run both server, we will know the query vectors μ_0, μ_1, and thus can know which row is being queried by simply adding them up (since they're bit vectors, it means XOR them.)
The flag: `CNS{c011u5i0n_a44ack_unav0idabl3}`

(c) & (d)

Cannot even understand...

6. Randomness Casino

- Reference: Some helper function from this article and this repo.

Please refer to `code/code6.py`. Helper functions are defined in `mt19937.py`.

1.

In this scenario, each player chooses a number that will be added up and take modular of 800, the resulting number will be the ID of winning player. Since we're the last player contributing to the sum, and we have known the sum so far, trivially we can choose a number such that we always win!
The flag: `CNS{f1N@L_con7RIbU7iON_47T@cK}`

2.

In this scenario, python's built-in random is used with a random seed, which is based on the MT19937 algorithm. Simply put, what MT19937 does is:

- Use seed to initialize 624 "states"
- The state is then "twisted" to get a new set of states
  - `twist()` can either be reversed with all 624 states known, or otherwise a certain number of states can be reversed given certain previous states.
  - The second case is being used here to reverse first 5 states.
- To get a random number, we `extract()` from states one by one
  - `extract()` is easily reversible without knowing other states
- With all 624 states extracted, the states are "twisted" again.

(continue on next page)

Here, our position is randomly generated and **we do not know the number for first 5 players (excluding us)**. However, we can deliberately give up the games until we're being placed after (and includes) 630th player. Under this condition, we can recover the choice of all 799 players, here are the steps:

Firstly, we can obtain the states from random numbers using the below function:

```python
def untemper(y):
    y ^= y >> mt19937.l
    y ^= y << mt19937.t & mt19937.c
    for i in range(7):
        y ^= y << mt19937.s & mt19937.b
    for i in range(3):
        y ^= y >> mt19937.u & mt19937.d
    return y
```

This function is independent for each random number i.e. no other states required to call `untemper()`.

As we mentioned, we can reverse the `twist()` action (which we call `backtrace()`) to obtain the initial 624 states (after initial twist) given previous states. The below function:

```python
def backtrace(cur, cnt):
    high = 0x80000000
    low = 0x7fffffff
    mask = 0x9908b0df
    state = cur
    for i in range(cnt,-1,-1):
        tmp = state[i+624]^state[i+397]
        if tmp & high == high:
            tmp ^= mask
            tmp <<= 1
            tmp |= 1
        else:
            tmp <<=1
        res = tmp&high
        tmp = state[i-1+624]^state[i+396]
        if tmp & high == high:
            tmp ^= mask
            tmp <<= 1
            tmp |= 1
        else:
            tmp <<=1
        res |= (tmp)&low
        state[i] = res
    return state
```

Will reverse the `twist()` action for first `cnt` of current states given initial 624-5 states (excluding first 5) Here we can take `cnt=5` and recover the initial 624 states, and it suffices to call `extract()` on them to know the first 624 player's choice.

As for the player after us, a simple `twist()` then `extract()` can let us obtain the next 624 choices (which is more than enough!) and thus we can decide the number to choose in order to let us win (by building the identical PRNG via setting identical initial states)

Although we deliberately give up on games, the expected value of our gain in each round is $(171/800) \times 800 - (631/800) \approx 170$. With sufficient trials we can eventually save up to 20000G and buy the flag.

The flag: `CNS{MT19937PredictorAlsoKnowsThePast!?}`.

Bonus

I didn't do the bonus, the code is just a naive brute force search, please ignore it.