CNS Homework 2

B11902083 徐粲桓

1. BGP

Reference: Discussed with B11902054 周得壹

(a)

Due to the Longest Prefix Match rule of BGP, the attacker can announce two subnets of prefixes longer by one digit and covers the subnet `10.10.220.0/22` originally belonged to AS1000. Since $11011100_2 = 220_{10}$ and the first 6 digits is the prefix, we can construct:

- `10.10.220.0/23 AS999`
- `10.10.222.0/23 AS999`

If the attack announce this rule, when a packet is being forwarded to `10.10.220.0/22`, it will match two rules: `10.10.220.0/22 AS1000` and one of `10.10.220.0/23 AS999`, `10.10.220.0/23 AS999`, where the one with longest prefix i.e. the route to `AS999` will be chosen, effectively hijacking traffic targeted for `10.10.220.0/22` under `AS1000`.

> *Note: The new description suggests that the attacker only want to target `10.10.220.XXX`, therefore,* ***adding `10.10.220.0/24 AS999` actually suffices.***

(b)

BGP blackholing can be used to mitigate a DDoS attack where one can temporarily redirect malicious traffic away from important services into a blackhole (a service that simply drops incoming traffic). This is more practical than attempting to block the host as DDoS attack usually comes from large number of hosts (e.g. Botnet)

(c)

The attacker can announce `{10.10.220.0/24, [AS999, AS2, AS1, AS1000]}` to effectively redirect the traffics just like in (a).

(d)

- Path prepending: The attacker `AS999` prepends not just itself but also the path `AS2, AS1, AS1000`, while normally it should only prepend itself and forward the update.
- Loop prevention: The attacker misuses this feature to decide "which" AS's traffic to redirect, as AS's will simply discard the path upon receiving the path containing themselves. Here, the attacker want to target `AS3, AS4, AS5` and ignore `AS1, AS2`, so they would announce a path containing `AS1, AS2, AS1000` to `AS1000` so that `AS1, AS2, AS1000` will not keep this path.

(e)

Generally, this technique is useful when one want to redirects high-volume of traffic via a path of their preference. For example, a company wanting to filter out malicious DDoS traffic by redirecting the traffic to a centralized data scrubbing center can use this technique to choose which AS they want to protect and from which AS the packets origin from to filter.

(f)

Advantage: Setting a MPL basically disables BGP prefix hijacking for rules that is already of the length of longest prefix as any attempts to advertise a longer prefix is impossible.

Disadvantage: A MPL causes a degradation in the granularity of routing as advertising AS path for smaller subnets than the limit is no longer available, which makes the protocol less convenient and flexible.

2. Internet Insecurity

Reference: All by myself.

(a)

The SYN flood attack puts a great load on the server's connection table as server need to keep a state for each connection in the traditional TCP Three-way handshake. SYN cookie, however, only require server to compute & verify SYN cookie per request while keeping no state (The cookie is sent back with SYN-ACK)

- A timestamp is needed for checking if the connection attempt is outdated.
- Client's IP address is to prevent stealing other's session: An eavesdropped SYN cookie without client's IP has no difference with other SYN cookies of same timestamp and MSS.

(b)

Yes. The ACK packet works as a "confirmation" of a connection sent from client to server. Therefore, by launching ACK flood attack, the attacker can cause denial of service by sending bodyless (only contains the ACK tag in TCP header) packets that do not belongs to any session in connection table, creating extra work for server to determine if the ACK packet is of a valid session.

(c)

Yes. Binding a valid SMTP server with its IPs and domain names cause issue since sometimes the sender's server run on dynamic IPs (provided by ISPs) or has a temporary domain name. An attacker can perform, say, domain hijacking to make their own service valid for the receiver.

3. Perfect Zero Knowledge

Reference: Discussed B11902054 周得壹 and B11902163 羅翊庭

(a)

Construction of V

WLOG, assume $|G_0| > |G_1|$.
Let the polynomial time deterministic verifier V takes a pair of graph (G_0, G_1) as an input and a transform matrix M of size $|G_0| \times |G_0|$ which represents the isomorphism π as the proof, and V essentially compute $\pi(G_0)$ by "multiplying" V_0 with the matrix and return $[[\pi(G_0) = G_1]]$. (Note: V will first check if $|G_0| = |G_1|$ and return 0 if not, otherwise, V will continue.)

To determine $[[\pi(G_0) = G_1]]$, V will compare the adjacent matrix of $\pi(G_0), G_1$ in $|G_0|^2$, where the adjacent matrix of $\pi(G_0)$ is computed from iterating the edges of G_0 in $|E_0| \leq |G_0|^2$ and rewrite the vertices in each edge with isomorphism π.

For example:

$$M = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, V_0 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, MV_0 = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

where MV_0 indicates that $\pi(1) = 3, \pi(2) = 2, \pi(3) = 1$. Suppose $G_0, G_1, \pi(G_0)$ has adjacent matrix $A_0, A_1, \pi(A_0)$:

$$A_0 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \pi(A_0) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

where $\pi(A_0)$ is computed from the zero matrix B, iterating $E_0 = \{(v_1, u_1), (v_2, u_2) \ldots\}$ and setting $B[\pi(v_i)][\pi(u_i)] = 1, \forall i$. V finally returns $[[\pi(A_0) = A_1]]$.

V is polynomial time since it performs $O(|G_0|^2) = O((|G_0| + |G_1|)^2)$ multiplications & comparison.

($\Leftarrow$) Soundness

If there's $x = (G_0, G_1)$ and a proof y such that $V(x, y) = 1$, then there's an isomorphism $y = \pi$ that maps G_0 to $\pi(G_0)$ and $\pi(G_0) = G_1$, then $G_0 \equiv G_1, x = (G_0, G_1) \in L$.

($\Rightarrow$) Completeness

If $x \in L$, $G_0 \equiv G_1, \exists \pi : \text{isomorphism}, \pi(G_0) = G_1, \pi \in \{0,1\}^{(|G_0|^2)}$ and $V(x, y = \pi) = 1$ since V returns $[[\pi(G_0) = G_1]]$.

(b)

Protocol A violates the perfect zero-knowledge property.

V^* Construction

We can simply let a cheating verifier such that it returns $b = 1$ in any round. Given that $G_0 \equiv G_1$ there exists a isomorphism $\pi, \pi(G_0) = G_1$ and V^* finds out π because $\pi_b = \pi$.

Other properties

As for completeness, the scheme is correct for honest P', V' as $id_n(G_0) = G_0, \pi(G_0) = G_1$ for $G_0 \equiv G_1$. Therefore, $\langle P', V' \rangle \langle G_0, G_1 \rangle = 1$.

For soundness, in one round a cheating prover P^* can fool the verifier with $1/2$ probability by sending $\pi_b = id_n$, but in the actual two-round protocol, the probability is $1/4$.

(c)

S^* Construction

Let S^* does the following:

- $\tau' \in_R S_n, b' \in_R \{0,1\}$
- $G := \tau'(G_{b'})$
- Sends G to V^* and receive challenge bit $b \in \{0,1\}$.
  - If $b \neq b'$, start over from first step.
  - Otherwise continue.
- Sends $\pi_b = \tau^{-1}$ and outputs whatever V^* does (Assuming V^* outputs its view)

Proof

To show that (P, V) is perfect zero-knowledge, we want to show that the distribution of $S^*(x) =$ the view of V^*, which includes G, b, π_b.

- G: τ', b' is randomly sampled and $G_1 = \pi(G_0)$, thus both $\tau'(G_1), \tau'(G_0)$ is uniformly distributed in S_n like $\tau(G_0)$.
- b: randomly sampled from $\{0, 1\}$ like b'.
- π_b
  - For $b = b' = 0$, it is trivial. $\pi_b = \tau^{-1}$ already in P.
  - For $b = b' = 1$, it's like P has chosen $\tau = \tau' \circ \pi$ and $\pi_b = \pi \circ (\tau' \circ \pi)^{-1} = \pi \circ \pi^{-1} \circ \tau^{-1} = \tau^{-1}$.

Since view of V^* from interaction with P is identical with the distribution of output of $S^*(x)$, Protocol 3 is perfect zero-knowledge.

(d)

A construction

Given the assumption, there exists expected polynomial time simulator S^* outputting view of V. Thus, we simply let the algorithm A do the following, given polynomial time constraint an^k:

- If A runs over the time constraint, it aborts and output 0.
- Otherwise, within the time constraint, A runs S^*.

$x \in L$

We let X be the random variable of S^* running time and $E[X] = f(n)$. By Markov's inequality:

$$P(X \geq an^k) = \frac{f(n)}{an^k} \to 0, k \to \infty$$

$$P(X < an^k) = 1 - \frac{f(n)}{an^k} \to 1, k \to \infty$$

By limit definition, $\forall \delta > 0, \exists N > 0, \forall k > N, P(X < an^k) \geq 1 - \delta$. In other words, given time limit cn^k. S^* can finish with at least probability $1 - \delta$. Therefore, completeness of A = Probability that S^* finish $\times$ completeness of S^* = $(1 - \delta)(1 - c)$.

We want to show that $Pr[A(x) = 1] \geq 1 - c - \epsilon, \forall \epsilon > 0$, so we can choose $\delta = 1 - (1 - c - \epsilon)/(1 - c)$ to satisfy that $Pr[A(x) = 1] = (1 - \delta)(1 - c) \geq 1 - c - \epsilon$.

Now, we want to find k such that $1 - f(n)/an^k \geq (1 - c - \epsilon)/(1 - c)$. We can choose

$$k = log_n(\frac{(1 - c)f(n)}{a\epsilon})$$

In conclusion, if we choose such k, we can have $Pr[A(x) = 1] \geq 1 - c - \epsilon, \forall \epsilon > 0$.

$x \notin L$

The definition of soundness error of P, V can be seen as the maximum probability for any P^* to fool V to output 1 for $x \notin L$, and our honest P is also included. Since S^* imitates the interaction of P, V, A should have same/less soundness error of P, V **when A do not exceed the time constraint**. Otherwise, A outputs 0 anyway.
Thus, suppose A has probability $p > 0$ to miss the time constraint, we have $Pr[A(x) = 1] \leq (1 - p)s \leq s$.

4. TLS

Reference: Mostly discussed B11902050 范綱佑 and B11902163 羅翊庭

By examining the packet content with Wireshark, it seems that the traffic consists of multiple TLSv1.2 session between a server `140.112.31.97` and a client `192.168.245.25`.

Since a TLSv1.2 session is established upon creation of a **master secret** which is derived from a **pre-master secret** generated by client & encrypted with server's public key, and it happens that the server's RSA modulus n is vulnerable to **Fermat's factorization**, which allows us to effectively crack the private exponent d.

Thus, one can generate the RSA private key with `Crypto.PublicKey` in Python and import as RSA key for TLS session in Wireshark `Preference → TLS`, which will decrypt the TLS traffic. (Refer to `code/code4.py`)

- If we try to log in with this key and the certificate of server, we will get rejected saying `You're role is not VIP! your role: TA`. Therefore, we want to forge a certificate with role `VIP`, which happens to be the client.

By observing the decrypted data we can retrieve the Root CA private key (say `root.key`) (in packet No. 185) , along with Root CA certificate, which can help us to **forge a certificate of a fake client** by pretending to be CA (Note: Root CA certificate is already included in the certificate chain inside the packet sent from server to client in `Server Hello Done` stage.)
This can be done in the following steps:

- Generate a public key for the fake client. This will be used for generating CSR (Certificate Signing Request) for CA to sign on: `openssl genrsa -out client.key 2048`
- Generate the CSR: `openssl req -new -key client.key -out client.csr`
- Process CSR: `openssl x509 -req -in client.csr -CA root.crt -CAkey root.key -CAcreateserial -out client.crt -days 365 -sha256`.
  - Reference

Using the signed certificate and private key for the fake client, we can login to the CNS System by `openssl s_client -connect cns.csie.org:4000 -cert client.crt -key client.key`. A username, password and a command will be prompted after login. Observing the decrypted traffic reveals several successful login attempts with valid command execution, where we can retrieve the credentials for getting the flag (specifically, the session of which Client Hello begins at packet No. 133)

- Username: `just_a_user`
- Password: `IL0VEw1Re5hark`
- Command: `giveMeTheFl4g`

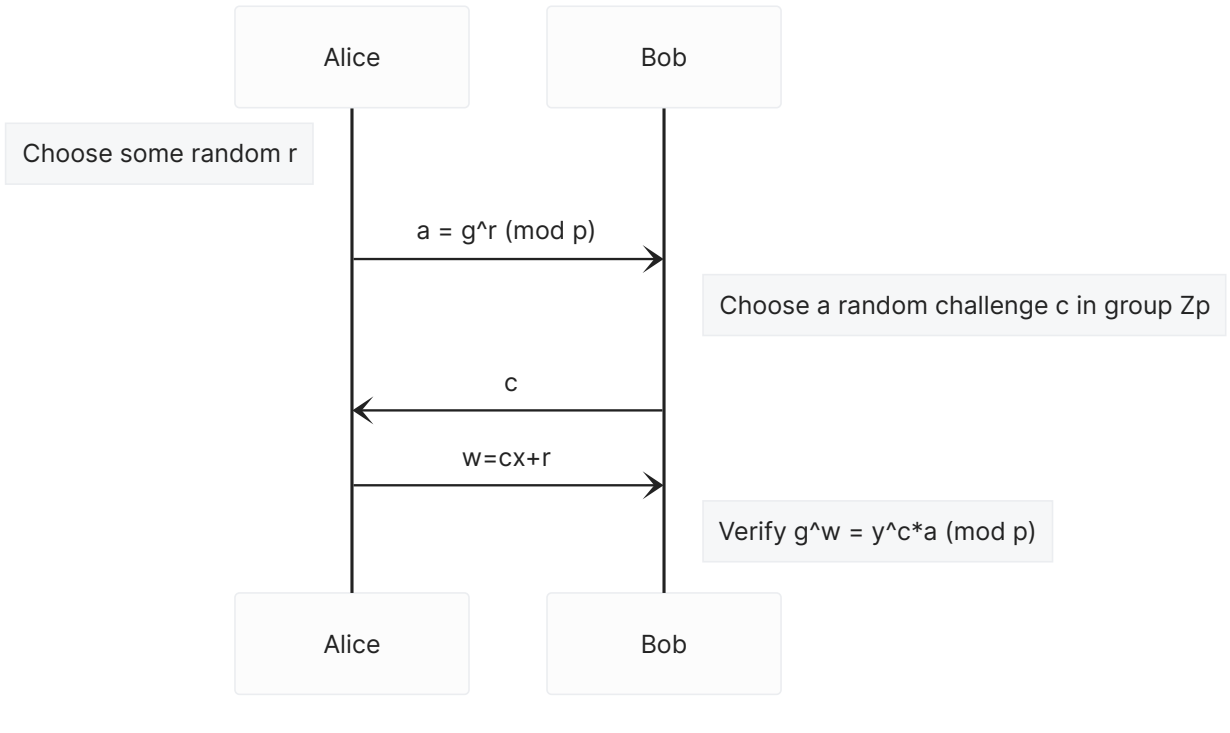
The flag: `CNS{TLS_4nD_w1re5h4rK_@rE_FuNNNNN!}`.

5. Ditto Knowledge Proof

Reference: All by myself.

(a)

Sequence diagram



Flag

Simply run `code5.py` and select sub-problem 1, the code will open two remotes (Alice, Bob) at the same time and performs MitM to make Bob believe that we're Alice.

The flag: `CNS{Y0U_1N_7H3_M1DDL3_4774CK_10c9d390f57b6b6f690519206ff3579b}`

(b)

First we analyze the parameters:

- What we have control on: a, w.
- c is generated from LCG, denote the generated c in i-th trial c_i.
  - Note that the LCG is initialized (i.e. called its `__init__` function) at the start of Bob's session. Therefore, between two trials of verifying Carrol, the LCG state is kept.
  - In `__init__` of LCG, the LCG always advances its state to the 87th step from its randomized initial state. Then for every trial, the step advances by 1.
  - Due to this, we can know c_2 based on c_1.
- y is known from `public.py`.

Note that our goal is simply give a pair of (a, w) such that $g^w = y^c \times a \bmod p$, consider the arithmetic in group Z_p^*:

- Let $w = 0$, we have $1 = y^c \times a$.
- $a = (y^c)^{-1}$, the modular inverse of y^c where y, c are known (in the second trial)

Providing the corresponding (a, w) pair gives the flag.

The flag: `CNS{P53UD0_R4ND0M_4R3_N07_R4ND0M_6ee07d69ce5587b46f638c90b341f43f}`

(c)

The system is insecure because it has no randomized nonce in the hash computation. If an adversary eavesdrops a pair of (a, w), they can do a replay attack by simply giving Bob the pair of (a, w) again and passes the test. To fix it, challenger need to let the prover show the "freshness" of their proof. For example, add a nonce to hash computation.

In particular, we can fix `verifier_non_interactive_mode()` to the following:

```
def verifier_non_interactive_mode(p: int, g: int, y: int):
    try:
        a = int(input('a = '))
        assert a > 0 and a < p
    except:
        print('Invalid input! You are weird! Bye!')
        exit(1)
    nonce = random.randint(0, sys.maxsize) # number in int range
    print('nonce =', nonce)
    c = H(nonce, p, g, y, a)

    try:
        w = int(input('w = '))
    except:
        print('Invalid input! You are weird! Bye!')
        exit(1)

    if pow(g, w, p) == (pow(y, c, p) * a) % p:
        return True
    else:
        return False
```

The original protocol still works as Admin can compute `c = H(nonce, p, g, y, a)` for `w`.

The flag: `CNS{R3PL4Y_4774CK_15_50000_51MPL3_a80e34ed1a90b3ae51755fc290dc81a8}`

(d)

If we can factorize the order $p - 1$ of multiplicative group of modulo p, we can use the Pohlig–Hellman algorithm to solve for private key i.e. solve for $x, g^x = y$ and derive message with $c_2 \times c_1^{-x}$. The idea of the algorithm is to "split" the group into subgroups, and apply baby-step giant-step algorithm to x in smaller groups and thus give simultaneous congruence. Finally, use CRT to find a solution. In particular:

- Input: Cyclic group G of order $n = \prod_{i=1}^{k} p_i^{e_i}$ with generator g and an element h.
- Output: $x \in G, g^x = h$.
- $\forall i = 1, \ldots, k$
  - $g_i := g^{n/p_i^{e_i}}$ is generator of subgroup of order $p_i^{e_i}$, by Lagrange's Theorem.
  - $h_i := h^{n/p_i^{e_i}} \in \langle g_i \rangle$ by construction and that g generates h.
  - Apply baby-step giant-step algorithm in $\langle g_i \rangle$ to find $x_i \in \langle g_i \rangle, g^{x_i} = h_i$.
- Solve simultaneous congruence: $x \equiv x_i (\bmod p_i^{e_i}), i = 1, \ldots, r$ with CRT
- Return x.

Fortunately, SageMath provides the function `discrete_log()` which implements Pohlig-Hellman, and we can simply call this method to solve for x and the flag. See `code5.py` for details.

The flag: `CNS{CDH_15_1N53CUR3_WH3N_50M37H1N6_F41L3D_f680114b0db7bd36f17b4c129c78f2de}`

6. So Anonymous, So Hidden

Reference: **(d)** inspired by B11902054 周得壹

(a)

Please refer to `code/code6a.py`.
The code basically accepts packet, decrypt it using private key and acquire next hop & raw packet content. Then, the packet is encoded and saved to a buffer of size 10. When the buffer is full, all the 10 packets is sent in randomized order to prevent timing analysis.

The flag: `CNS{Y0u_Ar3_A_g00D_m1x3R}`

(b)

Please refer to `code/code6b.py`.

The packet content consists of first 32 bytes being the stream cipher and the rest being the raw content. To add a hop, the target hop is prepended to raw content. Then, a random OTP is generated as the stream cipher key to encrypt raw content (with the target hop) and then OTP is encrypted with public key. To create a base packet, similar steps are taken but instead zero bytes are used as raw content.

After implementing `Packet.create()` and `add_next_hop()` we can create the base packet carrying the message with `create()` and adding the hops in reverse order (starting from the hop before Bob) by using the public key of previous hop (then the previous hop can decrypt and know which server to forward to.)

The flag: `CNS{b0X_1n_B0x_=_b0xC3pTi0n}`

(c)

Please refer to `code/code6c.py`.
Each RSA key pair of the servers has n small enough for factorization using SageMath. Given the first hop we can figure out the other hops by decrypting using private key of first hop (and so on.)

The flag: `CNS{https://youtu.be/jGqpz3stgzY}`

(d)

Please refer to `code/code6d.py`.

Using stem library we can derive the onion address `cns24otgzs5zmn2oztxnmiff7bscpg7kp7p6wnkayjnvr5fzxbqzeuqd.onion` from public key using `HideenServiceDescriptorV3.address_from_identity_key(pub_key)`. To find the port `11729` I used `proxychains` to scan all the ports of the address using `nmap -n -Pn -sT -p <ports> <address>` (subprocesses to speed up the scanning)

To interact with the Tor network, we need a SOCKS5 proxy that forwards our request (ref: ArchWiki - Tor) which can be done by enabling `tor.service`. Using `socks.set_default_proxy()` and acquire a socket using `socks.socksocket()`, we can get the server message asking us to solve a discrete log problem on singular curve (can be solved by using `singular_curve.py` from crypto-attacks) The solved k is of modular p', the order in the new multiplicative group after isomorphism in each round, and the actual flag can be solved using CRT. (Note: I chose a system of congruence of size 10 for CRT.)

Given that a group isomorphism will map a generator to generator, we can find the order of the multiplicative group with $u = \psi(P)$, `k = discrete_log(1/u, u)` and $p' = k + 1$ ($kP = 1/P \Rightarrow P \times (kP) = (k+1)P = O \Rightarrow [P]$ has order of $k+1$)

The flag: `CNS{y0u_kN0w_hW1_3_1_d}`