GlobalBoost Coding Hacks

Privacy and Security Hacks Compilation for GlobalBoost (BSTY)

Below, I've compiled all the relevant code snippets that focus on privacy and security features. These include embedding private messages, sending coins privately (e.g., via CoinJoin, stealth addresses, PayJoin), and other mechanisms like decoys, blind signatures, covenants, and more. I've kept them in their original numbering for reference, with a brief *Why* and *How* recap, followed by the code. This covers Taproot, SegWit, Schnorr, and related tech for BSTY. You can copy-paste directly.

1. Create Ordinals-Like Inscriptions for NFTs on BSTY Using Taproot (Private Data Embedding)

Why: Embeds data privately for immutable, hidden NFTs or messages. *How*: Encode and inscribe in Taproot witness.

```
python
import requests
import json

# Hack: Inscribe data on BSTY (simplified Ordinals-like)
def inscribe_bsty_nft(inscription_data, priv_key_hex):
    # Build raw tx with Taproot witness (placeholder; use full lib like bitcoinlib for prod)
    envelope = b'\x63\x3f' + inscription_data.encode()  # OP_IF OP_PUSH "ord" ...
    raw_tx = "01000000...witness with envelope..."  # Dummy raw tx hex

    # Sign (simplified)
    # ... (use ecdsa for Schnorr sign)

    # Broadcast
    url = "https://bstyexplorer.globalboost.info/api/tx/send"
    payload = {"rawtx": raw_tx}
    response = requests.post(url, json=payload)
    return response.json().get('txid') if response.status_code == 200 else None

txid = inscribe_bsty_nft("{\"p\":\"bsty-nft\",\"op\":\"mint\",\"name\":\"VetBadge\"}", "your_priv_hex")
print(f"Inscribed NFT txid: {txid}")
```

2. Mix BSTY Coins Using Taproot-Enabled CoinJoin in Python (Private Coin Sending)

Why: Obscures origins with aggregated signatures. *How*: Collect inputs, aggregate, mix outputs.

```
python
import ecdsa
import hashlib

# Hack: Simple Taproot CoinJoin
def coinjoin_mix(inputs, priv_keys, output_amount):
    # Aggregate pubkeys for MuSig
    vks = [ecdsa.SigningKey.from_string(bytes.fromhex(sk), curve=ecdsa.SECP256k1).verifying_key for sk in priv_keys]
    pubkeys = [b'\x02' + vk.pubkey.point.x().to_bytes(32, 'big') if vk.pubkey.point.y() % 2 == 0 else b'\x03' + vk.pubkey.point.x().to_bytes(32, 'big') for vk in vks]
    agg_pubkey = hashlib.sha256(b''.join(pubkeys)).digest()  # Simplified MuSig

    # Build tx with mixed outputs (equal amounts to break links)
    raw_tx = "01000000..."  # Inputs from participants, outputs to new addrs

    # Sign aggregated
    sighash = hashlib.sha256(raw_tx.encode()).digest()
    sigs = [sk.sign_digest(sighash, sigencode=ecdsa.util.sigencode_der) for sk in [ecdsa.SigningKey.from_string(bytes.fromhex(sk), curve=ecdsa.SECP256k1) for sk in priv_keys]]
    agg_sig = b''.join(sigs)  # Aggregate

    # Add to tx and broadcast
    return raw_tx + agg_sig.hex()  # Full tx hex for BSTY explorer

mixed_tx = coinjoin_mix(["input1", "input2"], ["priv1hex", "priv2hex"], 10000000)
print(mixed_tx)
```

3. Create Confidential Amounts with Taproot Pedersen Commitments in Rust (Private Value Hiding)

Why: Hides tx amounts provably. *How*: Commit and embed in Taproot.

```
rust
use secp256k1::{Secp256k1, Scalar, PublicKey};

// Hack: Confidential amount commitment
fn pedersen_commit(amount: u64, blinding: Scalar) -> PublicKey {
    let secp = Secp256k1::new();
    let g = PublicKey::from_slice(&[/* G generator */]).unwrap();
    let h = PublicKey::from_slice(&[/* H generator */]).unwrap();
```

```
    let commit = g.mul_tweak(&secp, &Scalar::from(amount)).unwrap()
        .combine(&h.mul_tweak(&secp, &blinding).unwrap()).unwrap();
    commit
}

fn embed_in_taproot(commit: PublicKey) -> String {
    // Tweak into Taproot output
    "gb1q...confidential_addr".to_string()  // Derive address
}

let blinding = Scalar::random();
let commit = pedersen_commit(100000000, blinding);
println!("Confidential Addr: {}", embed_in_taproot(commit));
```

4. Perform Privacy-Preserving Atomic Swaps with BTC Using Taproot HTLCs in JavaScript (Private Cross-Chain Sending)

Why: Secure swaps without intermediaries. *How*: Set up HTLC, claim privately.

```
javascript
const crypto = require('crypto');

// Hack: Taproot HTLC for atomic swap
function createHTLC(preimage, recipientPubkey, timeout) {
    const hash = crypto.createHash('sha256').update(preimage).digest('hex');
    // Script: OP_IF <hash> OP_EQUALVERIFY <recipient> OP_CHECKSIG OP_ELSE <timeout>
OP_CHECKLOCKTIMEVERIFY OP_DROP <refund> OP_CHECKSIG OP_ENDIF
    const taprootScript = `Taproot tree with HTLC branches`;  // Encode
    return { address: 'gb1q...htlc', hash };
}

function claimSwap(txid, preimage) {
    // Build spend tx revealing preimage
    return 'signed_tx_hex';  // Broadcast to BSTY
}

const preimage = 'secret';
const htlc = createHTLC(preimage, 'recipient_pub', 144);  // 24h timeout
console.log(htlc);
```

5. Build Hidden Payment Channels with Taproot for Off-Chain Privacy in Go (Private Off-Chain Sending)

Why: Scale private micro-payments. *How*: Fund channel, update states off-chain.

```
go
```

```
package main
import (
    "crypto/sha256"
    "fmt"
    "github.com/btcsuite/btcd/btcec"
)

// Hack: Taproot payment channel
func openChannel(pubkeys [][]byte, fundingAmount int64) string {
    // Aggregate pubkeys
    h := sha256.Sum256(append(pubkeys[0], pubkeys[1]...))
    return fmt.Sprintf("Channel Funding Tx: %x", h)  // P2TR address
}

func updateState(balanceA, balanceB int64) []byte {
    // New commitment tx (hidden in Taproot)
    return []byte("updated_state_sig")
}

func main() {
    channel = openChannel([][]byte{{/* pubA */}, {/* pubB */}}, 100000000)
    fmt.Println(channel)
}
```

6. Generate Decoy Outputs in SegWit Tx for Transaction Graph Obfuscation in Python (Private Tx Masking)

Why: Confuses ownership analysis. *How*: Add random outputs.

```
python
```

```
import random

# Hack: Add decoy outputs
def obfuscate_tx(inputs, real_output, amount, num_decoys=3):
    tx = {"inputs": inputs, "outputs": [(real_output, amount)]}
    for _ in range(num_decoys):
        decoy_addr = f"gb1q{random.randbytes(20).hex()}"
```

```
        decoy_amount = random.randint(1000, 10000)  # Dust
        tx["outputs"].append((decoy_addr, decoy_amount))

    # Adjust change, sign, etc.
    raw_tx = "segwit_tx_with_decoys"
    return raw_tx

obfuscated = obfuscate_tx(["input1"], "real_addr", 50000000)
print(obfuscated)
```

7. Use Taproot Key Tweaking for Stealth Addresses in Python (Private Receiving)

Why: One-time addresses prevent linkage. *How*: Tweak with shared secret.

```
python
import ecdsa
import hashlib
import bech32

# Hack: Generate stealth address
def generate_stealth_address(receiver_pubkey_hex, sender_priv_hex):
    receiver_pub = ecdsa.VerifyingKey.from_string(bytes.fromhex(receiver_pubkey_hex),
curve=ecdsa.SECP256k1)
    sender_sk = ecdsa.SigningKey.from_string(bytes.fromhex(sender_priv_hex),
curve=ecdsa.SECP256k1)
    shared_secret =
hashlib.sha256(receiver_pub.pubkey.point.mul(sender_sk.privkey.secret_multiplier).format()).digest()

    # Tweak receiver pubkey
    tweak = int.from_bytes(shared_secret, 'big')
    tweaked_point = receiver_pub.pubkey.point + ecdsa.ellipticcurve.Point(receiver_pub.pubkey.curve,
tweak, tweak)  # Simplified
    tweaked_pub = ecdsa.VerifyingKey.from_string(tweaked_point.format(compressed=True),
curve=ecdsa.SECP256k1)

    # Bech32 P2TR (gb prefix)
    x_only = tweaked_pub.to_string('compressed')[1:]
    address = bech32.encode('gb', bech32.convertbits(x_only, 8, 5))
    return address

stealth_addr = generate_stealth_address('receiver_pub_hex', 'sender_priv_hex')
print(stealth_addr)
```

8. Implement Ring Signatures with Schnorr for Anonymous Group Spending in JavaScript (Private Group Tx)

Why: Hides signer in group. *How*: Mix with decoys.

```javascript
const secp = require('@noble/secp256k1');

// Hack: Ring signature
function ringSign(message, realPrivKey, decoyPubKeys) {
    const realPubKey = secp.getPublicKey(realPrivKey);
    const ring = [realPubKey, ...decoyPubKeys];

    const e = secp.utils.sha256(message);  // Challenge
    const r = secp.utils.randomPrivateKey();  // Random
    const commitments = ring.map(pub => secp.getSharedSecret(r, pub));

    // Simplified ring (full impl needs Borromean or CLSAG for efficiency)
    const sig = secp.schnorr.sign(message, realPrivKey);
    return { sig, ring };  // Verify with aggregate check
}

const sig = ringSign('tx_hash', 'real_priv_hex', ['decoy1', 'decoy2']);
console.log(sig);
```

9. Hide Transaction Graphs with Decoy Inputs in SegWit Tx Using Go (Private Graph Obfuscation)

Why: Breaks input heuristics. *How*: Add unrelated inputs.

```go
package main
import (
    "fmt"
    "github.com/btcsuite/btcd/wire"
)

// Hack: Add decoy inputs
func addDecoyInputs(tx *wire.MsgTx, decoyUTXOs []string) {
    for _, utxo := range decoyUTXOs {
        // Parse txid:vout
        tx.AddTxIn(&wire.TxIn{PreviousOutPoint: wire.OutPoint{Hash: /* parse */}})
    }
```

```
    // Balance with OP_RETURN burn or recycle
}

func main() {
    tx := wire.NewMsgTx(wire.TxVersion)
    addDecoyInputs(tx, []string{"decoy_txid:0", "decoy_txid:1"})
    fmt.Printf("Obfuscated Tx: %v\n", tx)
}
```

10. Leverage Taproot for Zero-Knowledge Range Proofs in Rust (Private Value Proofs)

Why: Proves ranges without reveal. *How*: Embed in leaf.

```
rust
use bulletproofs::{BulletproofGens, PedersenGens, RangeProof};
use curve25519_dalek::scalar::Scalar;

// Hack: Range proof for confidential value
fn generate_range_proof(value: u64, blinding: Scalar) -> (RangeProof, /* commitment */) {
    let pc_gens = PedersenGens::default();
    let bp_gens = BulletproofGens::new(64, 1);
    let commitment = pc_gens.commit(Scalar::from(value), blinding);
    let proof = RangeProof::prove_single(&bp_gens, &pc_gens, &mut rand::thread_rng(), value, &blinding,
32).unwrap();
    (proof, commitment)
}

fn embed_in_taproot(proof: RangeProof) {
    // Add to Taproot script tree
}

let blinding = Scalar::random(&mut rand::thread_rng());
let (proof, _) = generate_range_proof(100000000, blinding);
```

11. Obfuscate Timestamps with Delayed Broadcasts and Taproot in Python (Private Timing)

Why: Breaks temporal correlations. *How*: Delay and broadcast.

```
python
import time
```

```
import random
import requests

# Hack: Delayed private broadcast
def delayed_broadcast(raw_tx, min_delay=60, max_delay=3600):
    delay = random.randint(min_delay, max_delay)
    time.sleep(delay)
    url = "https://bstyexplorer.globalboost.info/api/tx/send"
    response = requests.post(url, json={"rawtx": raw_tx})
    return response.json().get('txid')

# Assume signed Taproot tx
txid = delayed_broadcast("signed_tx_hex")
print(txid)
```

12. Deploy Private Smart-Like Contracts with Taproot Script Trees in Python (Private Conditions)

Why: Hides unused logic. *How*: Build merkle tree.

```
python
import ecdsa
import hashlib

# Hack: Taproot script tree for private conditions
def build_private_script_tree(conditions):
    leaves = [hashlib.sha256(cond.encode()).digest() for cond in conditions]  # Hash leaves
    while len(leaves) > 1:
        leaves = [hashlib.sha256(leaves[i] + leaves[i+1] if i+1 < len(leaves) else leaves[i]).digest() for i in
range(0, len(leaves), 2)]
    merkle_root = leaves[0]

    # Tweak internal key
    internal_pubkey = bytes.fromhex('internal_pub_hex')
    tweak = hashlib.tagged_hash("TapTweak", internal_pubkey + merkle_root).digest()
    tweaked_key = (int.from_bytes(internal_pubkey, 'big') + int.from_bytes(tweak, 'big')) %
ecdsa.SECP256k1.order

    # Output address (gb prefix)
    return f"gb1q{tweaked_key.to_bytes(32, 'big').hex()}"  # Simplified Bech32

addr = build_private_script_tree(["multi_sig_cond", "timelock_cond"])
print(addr)
```

13. Anonymize Inputs with PayJoin (P2EP) Using SegWit in JavaScript (Private Payments)

Why: Combines inputs to break heuristics. *How*: Collaborative tx building.

```javascript
const secp = require('@noble/secp256k1');

// Hack: PayJoin tx creation
function createPayJoin(payerInputs, payeeInput, outputs) {
    // Combine inputs
    const allInputs = [...payerInputs, payeeInput];

    // Reshuffle outputs to obfuscate
    outputs.sort(() => Math.random() - 0.5);

    // Sign collaboratively (SegWit witness)
    const sighash = secp.utils.sha256('tx_data');  // Placeholder
    const sigs = allInputs.map(input => secp.schnorr.sign(sighash, input.privKey));
    return { tx: 'combined_tx_hex', sigs };  // Broadcast
}

const pj = createPayJoin([{priv: 'payer_priv'}], {priv: 'payee_priv'}, [{addr: 'addr1', amt: 10000000}]);
console.log(pj);
```

14. Generate Ephemeral Keys for One-Time Signatures with Taproot in Go (Private Keys)

Why: Prevents reuse linkage. *How*: Derive and tweak.

```go
package main
import (
    "crypto/rand"
    "fmt"
    "github.com/btcsuite/btcd/btcec"
)

// Hack: Ephemeral Taproot key
func generateEphemeralTaproot(masterPriv []byte) string {
    ephemeralPriv := make([]byte, 32)
    rand.Read(ephemeralPriv)
    ephemeralPub, _ := btcec.PrivKeyFromBytes(btcec.S256(), ephemeralPriv).PubKey()
```

```
    // Tweak with master
    tweak := btcec.PrivKeyFromBytes(btcec.S256(), masterPriv).D.Bytes()
    tweakedPub := ephemeralPub.Add(tweak)  // Simplified

    return fmt.Sprintf("gb1q% x", tweakedPub.SerializeCompressed()[1:])  // X-only
}

func main() {
    addr := generateEphemeralTaproot([]byte("master_priv"))
    fmt.Println(addr)
}
```

15. Obscure Network Traffic with Tor Integration for BSTY Nodes in Rust (Private Networking)

Why: Hides IP for broadcasts. *How*: Route via Tor.

```
rust
use arti_client::TorClient;
use tokio::net::TcpStream;

// Hack: Broadcast tx over Tor
async fn broadcast_over_tor(raw_tx: &str) -> Result<String, Box<dyn std::error::Error>> {
    let tor = TorClient::create_bootstrapped().await?;
    let stream = tor.connect(("bsty_node_onion", 8333)).await?;

    // Send tx via RPC (simplified)
    let mut buf = vec![0u8; 1024];
    stream.read(&mut buf).await?;  // Response
    Ok(String::from_utf8(buf)?)
}

#[tokio::main]
async fn main() {
    let res = broadcast_over_tor("raw_tx_hex").await;
    println!("{:?}", res);
}
```

16. Apply Mimblewimble-Like Aggregation with Schnorr for Tx Compression in Python (Private Aggregation)

Why: Hides individual sigs. *How*: Aggregate batch.

```
python
import ecdsa

# Hack: Schnorr tx aggregation
def aggregate_tx_sigs(tx_list, priv_keys):
    sighashes = [hashlib.sha256(tx.encode()).digest() for tx in tx_list]
    sks = [ecdsa.SigningKey.from_string(bytes.fromhex(sk), curve=ecdsa.SECP256k1) for sk in priv_keys]

    agg_sig = b''
    for sk, sh in zip(sks, sighashes):
        sig = sk.sign_digest(sh, sigencode=ecdsa.util.sigencode_der)
        agg_sig += sig

    return agg_sig.hex()  # Compressed tx

agg = aggregate_tx_sigs(["tx1", "tx2"], ["priv1", "priv2"])
print(agg)
```

17. Utilize Schnorr Blind Signatures for Anonymous Credentials in Python (Private Credentials)

Why: Issues hidden credentials. *How*: Blind and unblind.

```
python
import ecdsa
import random

# Hack: Blind Schnorr signature
def blind_sign(issuer_sk_hex, blinded_message):
    sk = ecdsa.SigningKey.from_string(bytes.fromhex(issuer_sk_hex), curve=ecdsa.SECP256k1)
    blinded_sig = sk.sign_digest(blinded_message, sigencode=ecdsa.util.sigencode_der)
    return blinded_sig

def unblind_sig(blinded_sig, blinding_factor):
    # Simplified unblinding (adjust R and s)
    r, s = ecdsa.util.sigdecode_der(blinded_sig)
    unblinded_s = (s - blinding_factor) % ecdsa.SECP256k1.order
    return ecdsa.util.sigencode_der(r, unblinded_s)

message = b'credential_data'
blinding_factor = random.randint(1, ecdsa.SECP256k1.order - 1)
```

```
blinded_msg = (int.from_bytes(hashlib.sha256(message).digest(), 'big') + blinding_factor) %
ecdsa.SECP256k1.order
blinded_sig = blind_sign('issuer_priv_hex', blinded_msg.to_bytes(32, 'big'))
unblinded = unblind_sig(blinded_sig, blinding_factor)
print(unblinded.hex())
```

18. Create Privacy-Focused Covenant Scripts with Taproot in JavaScript (Private Spending Rules)

Why: Restricts spends privately. *How*: Hash and commit.

```
javascript
const crypto = require('crypto');

// Hack: Covenant script
function createCovenantScript(allowedAddr) {
    const covenantHash = crypto.createHash('sha256').update(allowedAddr).digest('hex');
    // Taproot leaf: OP_CHECKSIG OP_DROP OP_PUSH covenantHash OP_EQUAL
    const treeRoot = crypto.createHash('sha256').update(covenantHash).digest('hex');
    return `Tweaked output with covenant: ${treeRoot}`;
}

const covenant = createCovenantScript('allowed_addr');
console.log(covenant);
```

19. Mask Identities with Threshold Signatures via MuSig in Go (Private Group Auth)

Why: Hides participant count. *How*: Partial aggregate.

```
go
package main
import (
    "crypto/sha256"
    "fmt"
    "github.com/btcsuite/btcd/btcec"
)

// Hack: Threshold MuSig
func thresholdSign(privKeys [][]byte, message []byte, threshold int) []byte {
    if len(privKeys) < threshold {
        return nil
```

```
    }
    aggPub := []byte{}
    for _, pk := range privKeys[:threshold] {
        pub, _ := btcec.PrivKeyFromBytes(btcec.S256(), pk).PubKey()
        aggPub = append(aggPub, pub.SerializeCompressed()...)
    }
    aggHash := sha256.Sum256(aggPub)
    return aggHash[:]  # Simplified agg sig
}

func main() {
    sig := thresholdSign([][]byte{{/*priv1*/}, {/*priv2*/}, {/*priv3*/}}, []byte("msg"), 2)
    fmt.Printf("%x\n", sig)
}
```

20. Forge Private Audit Trails with Zero-Knowledge Set Membership in Rust (Private Proofs)

Why: Proves inclusion hidden. *How*: Use accumulators.

```
rust
use pairing::bls12_381::Bls12;
use zk_snark::Proof;

// Hack: ZK set membership
fn zk_membership_proof(set: Vec<u64>, element: u64) -> Proof<Bls12> {
    // Simplified: Use RSA accumulator or BLS
    // Accumulate set
    let acc = set.iter().fold(1u64, |a, &b| a * b);
    // Prove element divides acc without reveal
    Proof::dummy()  // Placeholder for real zk lib
}

let proof = zk_membership_proof(vec![1,2,3], 2);
```

21. Conceal Data Flows with Homomorphic Encryption in SegWit Outputs in Python (Private Computations)

Why: Encrypted ops on data. *How*: Aggregate ciphers.

```
python
from phe import paillier
```

```
# Hack: Homomorphic encryption
def homomorphic_add(enc1, enc2, pub_key):
    return enc1 * enc2  # Paillier multiplication = addition

pub, priv = paillier.generate_paillier_keypair()
enc1 = pub.encrypt(5)
enc2 = pub.encrypt(3)
sum_enc = homomorphic_add(enc1, enc2, pub)
print(priv.decrypt(sum_enc))  # 8
```

22. Embed Private Messages in BSTY Transactions Using Taproot (Private Messaging)

Why: Tamper-proof hidden notes. *How*: Commit and embed.

```
python
import ecdsa
import hashlib

# Hack: Embed private message in Taproot
def embed_private_message(message, recipient_pub_hex):
    encrypted_msg = hashlib.sha256(message.encode()).hex()  # Simple encrypt
    # Build Taproot tree with hidden leaf
    leaf_hash = hashlib.sha256(encrypted_msg.encode()).digest()
    # Tweak key (simplified)
    pub = bytes.fromhex(recipient_pub_hex)
    tweak = int.from_bytes(leaf_hash, 'big')
    tweaked = (int.from_bytes(pub, 'big') + tweak) % ecdsa.SECP256k1.order
    return tweaked.to_bytes(32, 'big').hex()  # Use in tx output

embedded = embed_private_message('Private note: Meeting at 5pm', 'recipient_pub')
print(embedded)
```