Analysis of `vulnTLSServer.nse`: TLS, CBC/SHA-1 Checks and Hello Handling

1 TLS, CBC and SHA-1 Handling in `vulnTLSServer.nse`

1.1 Negotiated Cipher Checks (CBC / SHA-1) in the Main Action

Once a TLS handshake succeeds, the script inspects the negotiated cipher and compression method. This logic lives in the `action` function, immediately after the protocol probes:

```
-- Negotiated CBC/SHA-1 or compression -> CRITICAL
if hello and hello.cipher then
  local name = cipher_name_from_field(hello.cipher)
  local U = tostring(name):upper()
  if U:find("CBC", 1, true) or U:find("SHA1", 1, true) or U:match("SHA$") then
    add_unique(critical_alerts, "Cipher includes CBC and/or SHA-1: " ..
        pretty_tls_name(name))
  end
end
if hello and hello.compression and tostring(hello.compression):upper() ~= "NULL"
     then
  add_unique(critical_alerts, "TLS compression is enabled (CRIME risk)")
end
```

Conceptually:

- **Location in file:** inside `action`, in the "CRITICAL" section just after the self-signed certificate check.
- **Role:** if the *negotiated* cipher suite contains `CBC`, `SHA1`, or ends in `SHA`, it raises a *CRITICAL* alert. Non-null compression also becomes *CRITICAL*.

1.2 Capability Probes for Known Bad CBC/SHA-1 Ciphers

The script also actively tests whether the server is willing to negotiate known-bad CBC/SHA-1 suites, even if they are not used by default. These ciphers are defined in the helper section:

```
-- Known-bad (CBC/SHA1) to test capability explicitly (CRITICAL per enunciado)
local ciphers_bad_cbc_sha1 = {
  "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
  "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
  "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA",
  "TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA",
  "TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
  "TLS_DHE_RSA_WITH_AES_256_CBC_SHA",
  "TLS_RSA_WITH_AES_128_CBC_SHA",
  "TLS_RSA_WITH_AES_256_CBC_SHA",
  "TLS_RSA_WITH_3DES_EDE_CBC_SHA",
}
```

The actual probe is performed later in `action`, in the “Capability probes” block:

```
-- === Capability probes (supported-but-not-negotiated) ====================
-- CBC/SHA acceptance must be CRITICAL per enunciado
if supports12 then
  local badHello, _ = tls_probe(host, port, "TLSv1.2", ciphers_bad_cbc_sha1, {"
      NULL"})
  if badHello and badHello.cipher then
    local cname = cipher_name_from_field(badHello.cipher)
    local U = tostring(cname):upper()
    if U:find("CBC", 1, true) or U:find("SHA1", 1, true) or U:match("SHA$") then
      add_unique(critical_alerts,
        "Server ACCEPTS CBC/SHA-1 cipher when offered: " .. pretty_tls_name(cname
            ))
      add_unique(high_alerts,
        "Server supports cipher outside allowlist: " .. pretty_tls_name(cname))
    end
  end
else
  if supports10 or supports11 then
    local badHello2, _ = tls_probe(host, port, "TLSv1.0", ciphers_bad_cbc_sha1,
        {"NULL"})
    if badHello2 and badHello2.cipher then
      local cname = cipher_name_from_field(badHello2.cipher)
      local U = tostring(cname):upper()
      if U:find("CBC", 1, true) or U:find("SHA1", 1, true) or U:match("SHA$")
          then
        add_unique(critical_alerts,
          "Server ACCEPTS CBC/SHA-1 cipher when offered (TLS 1.0/1.1): "
          .. pretty_tls_name(cname))
        add_unique(high_alerts,
          "Server supports cipher outside allowlist: " .. pretty_tls_name(cname))
      end
    end
  end
end
```

Conceptually:

- **Location in file:** helper table near the top, and a capability-probe section in `action`.
- **Role:** if the server accepts one of these CBC/SHA-1 suites when explicitly offered, the

script raises:

- *CRITICAL:* server accepts CBC/SHA-1.
- *HIGH:* server supports a cipher outside the allowlist.

1.3 Extended Sweep of Weak Ciphers

To go beyond the small list, the script performs a wider sweep of both weak and modern ciphers:

```
-- === Extended cipher sweep: RC4/3DES/CBC/SHA1 -> CRITICAL =================
do
  stdnse.debug1("vulnTLSServer: performing extended cipher probe list")
  local sweep = {
    "TLS_RSA_WITH_AES_128_CBC_SHA",
    "TLS_RSA_WITH_AES_256_CBC_SHA",
    "TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
    "TLS_DHE_RSA_WITH_AES_256_CBC_SHA",
    "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
    "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
    "TLS_RSA_WITH_3DES_EDE_CBC_SHA",
    "TLS_RSA_WITH_CAMELLIA_128_CBC_SHA",
    "TLS_RSA_WITH_CAMELLIA_256_CBC_SHA",
    "TLS_RSA_WITH_SEED_CBC_SHA",
    "TLS_RSA_WITH_RC4_128_SHA",
    "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",
    "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
    "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
    "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
    "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",
    "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256",
    "TLS_DHE_RSA_WITH_AES_128_GCM_SHA256",
    "TLS_DHE_RSA_WITH_AES_256_GCM_SHA384",
    "TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256",
    "TLS_RSA_WITH_AES_128_GCM_SHA256",
    "TLS_RSA_WITH_AES_256_GCM_SHA384",
    "TLS_RSA_WITH_AES_128_CBC_SHA256",
    "TLS_RSA_WITH_AES_256_CBC_SHA256"
  }

  for _, cs in ipairs(sweep) do
    local result, _ = tls_probe(host, port, "TLSv1.2", { cs }, {"NULL"})
    if result and result.cipher then
      local cname = cipher_name_from_field(result.cipher)
      local U = tostring(cname):upper()
      if U:find("RC4",1,true) or U:find("3DES",1,true)
         or U:find("CBC",1,true) or U:find("SHA1",1,true)
         or U:match("SHA$") then
        add_unique(critical_alerts,
          "Server ACCEPTS weak cipher: " .. pretty_tls_name(cname))
      elseif not (allowed_tls[cname] or allowed_lab[to_lab_style(cname)]) then
        add_unique(high_alerts,
          "Server supports cipher outside allowlist: " .. pretty_tls_name(cname))
      end
    end
  end
end
```

Conceptually:

- **Location in file:** within `action`, in the extended sweep block.
- **Role:** identify additional weak ciphers (RC4, 3DES, CBC, SHA-1) and raise *CRITICAL* if accepted.

2 ClientHello / ServerHello Handling

2.1 Literal Implementation of the TLS Probe

The function `tls_probe` is the core “Hello” engine. It constructs a TLS *ClientHello*, sends it, and then parses *ServerHello* (or alerts) using Nmap’s `tls` library:

```lua
-- Minimal TLS probe
local function tls_probe(host, port, protocol, ciphers, compressors, named_curve
    )
  local timeout_ms = (((host.times and host.times.timeout) or 5) * 1000) + 5000
  local sock = nmap.new_socket()
  sock:set_timeout(timeout_ms)
  local ok, err = sock:connect(host, port)
  if not ok then sock:close(); return nil, ("connect failed: %s"):format(err or
      "unknown") end

  local t = {
    protocol = protocol or "TLSv1.2",
    record_protocol = protocol or "TLSv1.2",
    ciphers = ciphers or ciphers_modern_tls12,
    compressors = compressors or {"NULL"},
    named_curve = named_curve
  }

  local req = tls.client_hello(t)
  ok, err = sock:send(req)
  if not ok then sock:close(); return nil, ("send failed: %s"):format(err or "
      unknown") end

  local buffer, i = "", 1
  local function next_record()
    local idx, rec = tls.record_read(buffer, i)
    if rec == nil then
      local okb, buf, e = tls.record_buffer(sock, buffer, i)
      if not okb then return nil, e end
      buffer, i = buf, 1
      idx, rec = tls.record_read(buffer, i)
      if rec == nil then return nil, "no more records" end
    end
    i = idx
    return rec
  end

  local negotiated = { version = nil, cipher = nil, compression = nil }
  while true do
    local rec, e = next_record()
    if not rec then sock:close(); return negotiated, e end
    if rec.type == "alert" then sock:close(); return negotiated, "alert" end
    if rec.type == "handshake" and rec.body then
      for _, msg in ipairs(rec.body) do
        if msg.type == "server_hello" then
          negotiated.version = msg.protocol or protocol
          negotiated.cipher = msg.cipher
          negotiated.compression = msg.compressor or msg.compression or "NULL"
          sock:close()
          return negotiated, nil
        elseif msg.type == "server_hello_done" then
          sock:close()
          return negotiated, nil
        end
      end
    end
  end
end
```

Conceptually:

- **Literal location:** helper section near the top of the file, after the cipher lists.
- **Conceptual role:** send a ClientHello with specified version, cipher list, compression, and optional curve; then parse ServerHello to learn which version/cipher/compression the server chose.

2.2 Use of `tls_probe` Inside `action`

Protocol support probes. The script calls `tls_probe` early in `action` to discover which protocol versions are supported:

```
-- 2) Protocol support probes (avoid TLS1.3 crafting on older Nmap)
local hello12 = tls_probe(host, port, "TLSv1.2", ciphers_modern_tls12, {"NULL"})
local hello13 = nil
local hello11 = tls_probe(host, port, "TLSv1.1", ciphers_tls10_11, {"NULL"})
local hello10 = tls_probe(host, port, "TLSv1.0", ciphers_tls10_11, {"NULL"})

local supports12 = hello12 and (hello12.version and tostring(hello12.version):
    match("1%.2"))
local supports13 = false
local supports11 = hello11 and (hello11.version and tostring(hello11.version):
    match("1%.1"))
local supports10 = hello10 and (hello10.version and tostring(hello10.version):
    match("1%.0"))

if not (supports12 or supports13 or supports11 or supports10) then
  add_unique(high_alerts, "No TLS handshake completed on this port (likely not a
       TLS service)")
end

local hello = hello12 or hello11 or hello10
if hello then
  stdnse.debug1("vulnTLSServer: negotiated version=%s cipher=%s compression=%s",
    tostring(hello.version), tostring(hello.cipher), tostring(hello.compression))
end
```

Capability probes (CBC/SHA-1 and disallowed ciphers). The same `tls_probe` is reused for:

- Testing CBC/SHA-1 suites (`ciphers_bad_cbc_sha1`, see previous section).
- Testing explicitly disallowed TLS 1.2 suites:

```
if supports12 then
  local cmpHello, _ = tls_probe(host, port, "TLSv1.2",
                              ciphers_modern_tls12, {"NULL","DEFLATE"})
  if cmpHello and cmpHello.compression
     and tostring(cmpHello.compression):upper() ~= "NULL" then
    add_unique(critical_alerts,
      "Server supports TLS compression (selected " .. tostring(cmpHello.
          compression) .. ")")
  end

  local disallowed_probe = {
    "TLS_RSA_WITH_AES_128_GCM_SHA256",
    "TLS_RSA_WITH_AES_256_GCM_SHA384",
    "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256",
  }
  for _, cs in ipairs(disallowed_probe) do
    local h, _ = tls_probe(host, port, "TLSv1.2", { cs }, {"NULL"})
    if h and h.cipher then
      local cname = cipher_name_from_field(h.cipher)
      if cname then
        add_unique(high_alerts,
          "Server supports cipher outside allowlist: " .. pretty_tls_name(cname))
      end
    end
  end
end
```

Extended sweep and cipher preference. Finally, `tls_probe` is the engine both for the extended sweep and for the cipher-preference test:

```
-- Cipher preference test
do
  stdnse.debug1("vulnTLSServer: checking cipher preference (client vs server)")
  local reverse_ciphers = {
    "TLS_RSA_WITH_AES_256_GCM_SHA384",
    "TLS_DHE_RSA_WITH_AES_128_GCM_SHA256",
    "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256"
  }
  local negotiated, _ = tls_probe(host, port, "TLSv1.2", reverse_ciphers, {"NULL
      "})
  if negotiated and negotiated.cipher then
    local cname = cipher_name_from_field(negotiated.cipher)
    if cname then
      if cname == reverse_ciphers[1] then
        add_unique(low_alerts,
          "Cipher Preference: Server allows client-chosen cipher (client
              preference detected)")
      else
        stdnse.debug1("Server cipher preference enforced: %s", cname)
      end
    end
  else
    stdnse.debug1("Could not negotiate cipher preference test handshake")
  end
end
```

3 Environment-Feasible Tests and Observed Results

In the provided Docker/Apache environment (TLS 1.2-capable, no TLS compression, TLS 1.3-only mode not working reliably), the following tests have been executed and are realistically testable.

3.1 FP1: Modern TLS 1.2, No CBC/SHA-1, No False Positives

Goal. Ensure that with a "clean" TLS 1.2 configuration using only modern AEAD ciphers (for example ECDHE-RSA-AES256-GCM-SHA384) and allowed curves (X25519), the script does *not* raise CBC/SHA-related CRITICALs or TLS-support HIGHs.

Key configuration. Apache configured with:

- Protocol: TLS 1.2 only.
- Cipher: subset of the allowlist (GCM/CHACHA20, no CBC, no SHA-1).
- Curve: X25519 (as shown by `openssl s_client`).
- Self-signed certificate for `CN=localhost`.

```
sudo nmap -p 8443 --script ~/Desktop/lab1/vulnTLSServer.nse \
  -oN test/testFP1.out localhost
```

Command (Kali, in `~/Desktop/lab1`).

Relevant output (summary).

- **CRITICAL:** only “Self-signed certificate detected”.
- **HIGH:** “Certificate not yet valid: notBefore ... is in the future” (time-zone edge case) and “HSTS header not configured (missing Strict-Transport-Security)”.
- **MEDIUM:** “Sensitive information disclosed in header ’server: Apache/2.4.54 (Debian)”’.
- **LOW:** “Non-qualified hostname in certificate CN: localhost”, “Certificate missing SAN extension”.

Importantly:

- No “Cipher includes CBC and/or SHA-1”.
- No “Server ACCEPTS weak cipher” CRITICALs.
- No “No TLS handshake completed” or “Server does not support TLS 1.2 or TLS 1.3 by default”.

This validates that, in this environment, the CBC/SHA and TLS-support logic does not generate false positives for a modern TLS configuration.

3.2 CBC/SHA-1 Positive Tests

In separate tests, Apache was configured with CBC and SHA-1 based suites such as:

- `AES128-SHA`, `DES-CBC3-SHA`, and other CBC/SHA-1 ciphers from `ciphers_bad_cbc_sha1`.

The script correctly raised:

- CRITICAL alerts for “Cipher includes CBC and/or SHA-1: . . . ” and “Server ACCEPTS weak cipher: . . . ”.
- HIGH alerts for “Server supports cipher outside allowlist: . . . ”.

These tests confirm the positive detection paths for CBC/SHA-1 and weak ciphers.

3.3 TLS Version Handling

With Apache forced to only allow TLS 1.0/1.1, the script’s version probes using `tls_probe` produced:

- HIGH alerts such as “Server does not support TLS 1.2 or TLS 1.3 by default; obsolete protocols enabled: TLS 1.0 TLS 1.1”.

This validates that protocol version support is correctly inferred from the ClientHello/ServerHello exchange.

3.4 Non-Testable Case in This Environment: TLS 1.3-only

An attempt was made to configure Apache with TLS 1.3-only and TLS 1.3 cipher suites. However,

- `openssl s_client -tls1_3` to `localhost:8443` returned: “no peer certificate available”, cipher `(NONE)`.
- The TLS 1.3 handshake did not complete even at the OpenSSL level.

Therefore, behaviour for a strictly TLS 1.3-only server cannot be validated in this Docker environment, and potential edge cases involving TLS 1.3-only stacks remain out of scope for this lab.