Chapter 13

PASSIVE SCANNING IN MODBUS NETWORKS

Jesus Gonzalez and Mauricio Papa

Abstract This paper describes the design and implementation of a passive scanner for Modbus networks. The tool integrates packet parsing and passive scanning functionality to interpret Modbus transactions and provide accurate network representations. In particular, the scanner monitors Modbus messages to maintain and update state table entries associated with field devices. Entries in the state tables record important information including function codes, transaction state, memory access and memory contents. The performance and reporting capabilities of the passive scanner make it an attractive network troubleshooting and security tool for process control environments.

Keywords: Process control systems, Modbus protocol, passive network scanning

1. Introduction

Industrial processes are increasingly relying on sophisticated process control systems (PCSs) – also known as SCADA systems – for supervisory control and data acquisition. PCSs control industrial processes using networks of sensors and actuators. Sensors provide data about process variables as input to the PCS. Actuators make adjustments to the process variables based on output signals received from the PCS. The PCS control algorithm defines how PCS inputs are used to compute the output signals that drive the industrial process to the desired state.

In many industrial environments, sensors, actuators and controllers are deployed in widely dispersed locations, requiring a communication infrastructure and protocols to support supervisory control and data acquisition. The communication protocols have traditionally favored operational requirements over security because the field equipment and communications infrastructure were physically and logically isolated from other networks. However, the specifications for most major industrial protocols, e.g., Modbus [7, 8] and DNP3 [15, 16],

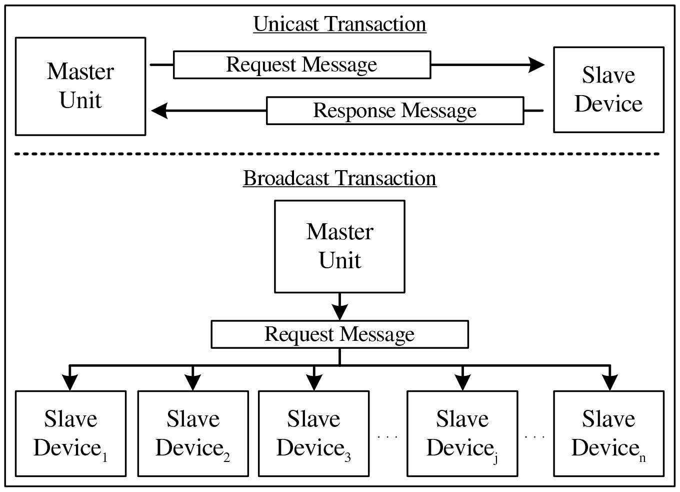

Figure 1. Master-slave transaction.

now include mechanisms to transport control data using TCP/IP stacks. The use of TCP/IP as a transport mechanism in industrial control networks raises serious issues, largely because it promotes the trend to interconnect with corporate IT networks. It is, therefore, extremely important to deploy security tools that are designed specifically for industrial control networks that use TCP/IP stacks [1, 3–5, 12].

This paper describes the design and implementation of a passive scanning tool for Modbus networks, which are commonly used for pipeline operations in the oil and gas sector. The scanning tool monitors Modbus protocol communications to obtain detailed information about network topology and control device configurations and status. As such, the tool is valuable to security administrators for event logging, troubleshooting, intrusion detection and forensic investigations [6, 13, 14].

2. Modbus Protocol

Modbus is a communication protocol for industrial control systems. Developed by Modicon (now Schneider Automation) in 1979, the Modbus protocol specifications and standards are currently maintained by the independent group Modbus IDA. Three documents are at the core of the Modbus standard: the protocol specification [7] and implementation guides for use in serial lines [10] and TCP/IP networks [8].

2.1 Function Codes

The Modbus protocol was designed as a simple request/reply communication mechanism between a master unit and slave devices. Figure 1 illustrates a Modbus transaction. Communication may occur over serial lines or, more recently,

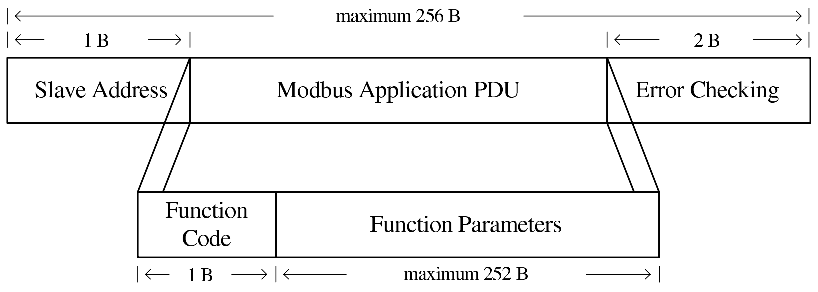

Figure 2. Modbus serial message.

using TCP/IP as a transport mechanism for Modbus messages. A function code included in a Modbus message describes the purpose of the message.

The simple, yet functional, structure of the Modbus protocol has contributed to its implementation by several vendors, enabling seamless interconnection of devices in multiplatform environments. The open nature of the Modbus specifications has led to its current standing as the *de facto* industry standard for process control system communications in the oil and gas sector.

Modbus messages have two major parts, a header with address and control information (possibly spanning multiple network layers) and a protocol data unit (PDU) specifying application-level operations. When the protocol is used in serial lines, messages also include error checking data as a trailer (Figure 2). PDUs comprise two fields [7]: (i) a function code part describing the purpose of the message, and (ii) a function parameters part associated with function invocation (for a request message) or function results (for a reply message). The function code is important because it specifies the operation requested by the master unit; also, it conveys error information in cases where an exception has occurred in a slave device.

The Modbus function code length is 8 bits and the maximum length of the PDU is 253 bytes, providing a maximum of 252 bytes for use as function parameters (Figure 2). The limit on PDU length originates from legacy implementations of Modbus on serial lines.

Modbus has three types of function codes: (i) public codes, (ii) user-defined codes, and (iii) reserved codes. Public codes correspond to functions whose semantics are completely defined or will be defined in the standard. Public code values fall in the non-contiguous ranges {1–64, 73–99, 111–127}. User-defined codes in the ranges {65–72, 100–110} support functions that are not considered in the standard. The implementation of user-defined codes is left to the vendor and there are no guarantees of functional compatibility in heterogeneous environments. Reserved function codes overlap with the space assigned to public codes; they correspond to public codes that are not available for public use to ensure compatibility with legacy systems [7]. Function codes in the range {128–255} are used to denote error conditions. If an error condition occurs for a function code $x \in \{1\text{–}127\}$ in a request from a master to a slave, the error

Table 1. Public codes for diagnostic functions.

Function	**Code**
Read Exception Status	7
Diagnostic	8
Get Communication Event Counter	11
Get Communication Event Log	12
Report Slave ID/Status	17
Report Device Identification	43

situation is indicated by the function code $x + 128$ in the reply message from the slave to the master.

Public function codes are used for diagnostics, and for data access and manipulation in slave devices. Most diagnostic codes are defined for obtaining status information from slave devices in serial lines (Table 1).

Data access functions are designed to read/write data objects from/to primary tables. Modbus defines four types of primary tables: discrete inputs, coils, input registers and holding registers. The first two types of tables contain single-bit objects that are read-only and read-write, respectively. The other two types of tables contain 16-bit objects that are read-only and read-write, respectively. Table 2 summarizes the public codes corresponding to data access functions for the four types of primary tables.

Single-bit discrete inputs and coils are normally associated with discrete I/O systems. On the other hand, input registers and holding registers are generally associated with analog systems, i.e., analog inputs and outputs, respectively. Any of the four primary data objects may also be used as program variables for implementing control logic.

The Modbus standard also permits files to be used to read and write device data (process-related data and configuration data). In this case, the data objects are called "records" and each file contains 10,000 records. The length of a record is file-dependent and is specified using a 16-bit word. It is important to note that the Modbus standard allows memory allocations for primary tables and files to overlap.

2.2 Transactions

A Modbus transaction involves the exchange of a request message from a master and a reply message from the addressed slave (except for broadcast messages, which have no reply messages). The master communicates using the unicast address associated with the slave device (i.e., its network ID) as the destination address of the request, or by sending a request message to the broadcast address [10, 11]. Note that if the broadcast address is used by the master, the request is received and processed by all listening slaves, but no response messages are provided by the slaves. When the unicast address of a

Table 2. Public codes for data access functions.

Function	**Code**	**Type**	**Size (Bits)**
Read Discrete Inputs	2	Read Only	1
Read Coils	1	Read/Write	1
Write Single Coil	5	Read/Write	1
Write Multiple Coils	15	Read/Write	1
Read Input Registers	4	Read Only	16
Write Single Register	6	Read/Write	16
Read Holding Registers	3	Read/Write	16
Write Multiple Registers	16	Read/Write	16
Read File Records	20	Read/Write	16
Write File Records	21	Read/Write	16
Mask Write Register	22	Read/Write	16
Read/Write Multiple Registers	23	Read/Write	16
Read FIFO Queue	24	Read/Write	16

slave is used, the addressed slave is required to send a response message back to the master.

Frame headers for Modbus messages in serial lines (Figure 2) only include the address of the intended slave recipient (for requests and replies). In a request message, this address identifies the recipient; in a response message, the address is used by the master to identify the responding slave. The address field is 8 bits long. The broadcast address is 0 (zero); values in the range {1–247} are used for individual slave addresses and values in the range {248–255} are reserved. Note that the maximum size of a Modbus frame in serial line is 256 bytes (including two trailer bytes used for error detection).

As described above, an error condition is indicated by sending a different function code in the reply message. Also, an exception code is included in the function parameter section of the PDU.

2.3 TCP/IP Services

Modbus TCP transactions are functionally equivalent to those specified in the serial version, i.e., master and slave devices exchange PDUs, except that transactions are encapsulated in TCP messages. Modbus TCP extends the functionality offered by the serial version by enabling slave devices to engage in concurrent communications with more than one master. Also, the master can have multiple outstanding transactions.

The implementation guide for Modbus messaging over TCP/IP [8] specifies that slave devices must listen for incoming TCP connections on port 502 (IANA assigned port) and may optionally listen on additional ports. The slave device that performs the passive open operation on TCP is designated as the "server." On the other hand, the master device that performs the active open operation on TCP is designated as the "client." Note that Modbus roles cannot

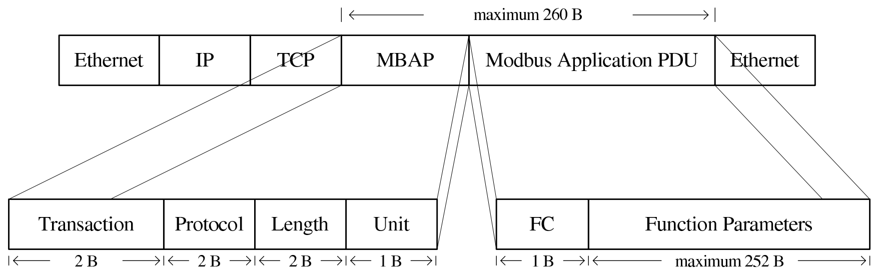

Figure 3. Modbus TCP message.

be changed on a TCP communication channel once it is established; however, multiple outstanding transactions may exist on the channel. A new communication channel is established when a device needs to assume a different role.

A Modbus TCP PDU includes an extra header to handle the additional capabilities. This Modbus Application Protocol (MBAP) header has four fields (Figure 3): (i) transaction identifier, (ii) protocol identifier, (iii) length, and (iv) unit identifier. The transaction identifier enables a device to pair matching requests and replies belonging to the same transaction. The protocol identifier indicates what application protocol is encapsulated by the MBAP header (zero for Modbus). The length field indicates the length in bytes of the remaining fields (unit identifier and PDU). Finally, the unit identifier indicates the slave device associated with the transaction.

The Modbus TCP specification requires that only one application PDU be transported in the payload of a TCP packet [8]. Since application PDUs have a maximum size of 253 bytes (see Figure 2) and the length of the MBAP is fixed at seven bytes, the maximum length of a Modbus TCP data unit is 260 bytes.

3. Architecture

This section describes the architecture of the passive Modbus scanner that monitors Modbus messages to identify and extract information about master and slave devices in an industrial control network. The information is useful for monitoring network status and troubleshooting device configurations and connections. Given an appropriate amount of time, the tool can discover each communicating Modbus device and the set of function codes used on the device. Also, the tool can monitor the status of Modbus transactions, i.e., whether they result in positive or negative responses.

To support security evaluations, the passive scanner can be used to detect and log the presence of rogue devices, monitor memory transfer operations and detect variations in network use in real time. Note that the monitoring and logging of memory transfer operations help detect anomalous activity and support forensic investigations.

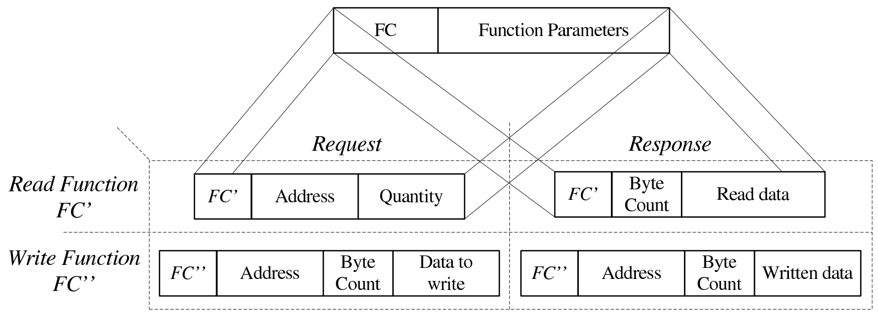

Figure 4. Modbus message structure for memory access operations.

3.1 Design and Features

The passive scanning tool has three main components: (i) a network scanner, (ii) a Modbus transaction checker, and (iii) an incremental network mapper. The network scanner passively captures and parses Modbus messages. Information captured by the scanner is passed to the transaction checker to pair matching sets of messages. The incremental network mapper uses the collected information to populate dynamic data structures that store network topology and status information.

Messages in a SCADA network tend to follow repetitive, often predictable communication patterns. Based on this assumption, we use an incremental network mapping algorithm that only updates its data structures when a new pattern, feature or device is identified. In most cases, the rate at which new information is added decreases over time; depending on the network traffic, no new information may be added for a relatively long period of time. Abrupt changes to the expected trend may indicate anomalous activity and could be used as a metric for anomaly detection.

Data access messages (Figure 4) provide valuable information associated with a slave device such as memory addresses, types and contents. Furthermore, since function parameters in Modbus messages always contain slave device information, i.e., a message is a request directed at a slave or a response from a slave, Modbus communications tend to reveal more information about slave devices than the master unit. Consequently, the reports produced by the passive scanner concentrate mainly on slave devices.

Whenever a Modbus message captured by the scanner is matched by the transaction checker, the network mapper inspects the transaction and updates the state of the data structure. The following information is extracted and stored: master id, slave id, function code and transaction status. In addition, for each operation that involves memory manipulation, the data type associated with the operation, access type (read/write), memory contents (accessed from or written to the slave device), and memory addresses associated with the data transfer are recorded.

Algorithm 1 : Passive Network Scanning.

```
Input: Network traffic
Output: Queue of Modbus messages ($MQueue$)
  Process command line options
  Set exit condition
  while exit condition not satisfied do
    $m \leftarrow$ packet captured from network
    if $m$ is $Modbus$ then
      $MQueue.add(m)$
    end if
  end while
```

3.2 Algorithms

This section describes the algorithms used for passive scanning of Modbus networks. Data input is provided by a simple packet capture utility that parses and filters Modbus messages (Algorithm 1). The algorithms use three key data structures:

- *MQueue*: This FIFO queue stores unprocessed Modbus messages. The network scanner inserts elements in the queue and the transaction checker removes elements from the queue during processing.

- *MTReq*: This hash table stores pending Modbus requests. Hash value i for $MTReq[i]$ is computed from Modbus message fields, i.e., $i = hash(x_1, x_2, \cdots, x_n)$. For example, a hash value could be computed using $x_1 = slaveIP$, $x_2 = masterIP$, $x_3 = slavePort$, $x_4 = masterPort$, and $x_5 = TransactionID$. Note that slave devices listen on TCP port 502 by default, thus, $x_3 = 502$ in most cases. The hash table is primarily used by the transaction checker to generate matching request and reply messages for Modbus transactions, which enable the incremental network mapper to collect device information.

- *MDev*: This hash table stores objects associated with a specific Modbus device. Hash value j for $MDev[j]$ is computed from Modbus message fields, i.e., $j = hash(y_1, y_2, \cdots, y_m)$. In this case, the set of fields uniquely identify a Modbus device on the network, i.e., $y_1 = slaveIP$, $y_2 = slave\ MAC\ address$, $y_3 = unitID$. Individual entries in the table store device-specific information such as the addresses associated with the slave device, addresses of master devices that communicated with the slave device, and the function codes and parameters seen in transactions associated with the device.

The transaction checker is responsible for matching request and reply Modbus messages that belong to the same transaction, i.e., it is a stateful transaction monitor (Algorithm 2). Note that only request messages are stored in

Algorithm 2 : Modbus Transaction Checking.

```
Input: Queue of unprocessed Modbus messages ($MQueue$)
Output: Queue of Modbus transactions ($TQueue$)
  while exit condition not satisfied do
    $m \leftarrow MQueue.next()$
    $i \leftarrow hash(x_1, x_2, \cdots, x_n)$
    if $m$ is a request then
      if $MTReq[i] \neq \emptyset$ then
        $log$("Multiple identical requests")
      end if
      $MTReq[i] \leftarrow m$
    else // Message $m$ is a response
      if $MTReq[i] = \emptyset$ then
        $log$("No matching request found")
      else
        $valid\_transaction = check\_transaction(MTReq[i], m)$
        if $valid\_transaction = false$ then
          $log$("Invalid transaction - (invalid request/reply parameters)")
        else // $MTReq[i]$ and $m$ constitute a valid transaction
          $TQueue.add(\{MTReq[i], m\})$
          $MTReq[i] \leftarrow \emptyset$
        end if
      end if
    end if
  end while
```

MTReq. Reply messages for which there are no matching requests are immediately logged to indicate the anomaly and *MTReq* remains unchanged. If a matching reply is received and validated, an entry is added to the transaction queue *TQueue* (an auxiliary data structure) and the request is removed from *MTReq*. On the other hand, if a matching reply is not validated, a log entry is produced and the request message remains in *MTReq* in case a matching reply is received later.

Transactions in *TQueue* are the input for the incremental network mapping algorithm (Algorithm 3). This algorithm records information about all Modbus devices *MDev* detected in network communications. First, the algorithm determines whether a transaction involves a new Modbus device; if this is the case, an entry associated with the new device is added to *MDev*. Next, the algorithm stores relevant transaction information, including the master ID, function codes and function parameters. Note that transaction information is considered relevant (and logged) only if it provides new information about the network.

Modbus transactions may involve memory access/updates on the remote devices as well as diagnostic operations. When operations manipulate device

Algorithm 3 : Incremental Network Mapping.

```
Input: Queue of Modbus transactions ($TQueue$)
Output: Hash table of Modbus device objects ($MDev$)
  while exit condition not satisfied do
    $trans \leftarrow TQueue.next()$
    $j \leftarrow hash(y_1, y_2, \cdots, y_m)$
    if $MDev[j] = \emptyset$ then // Handle new observed devices
      $MDev[j] \leftarrow newMDev(trans.slave)$
    end if
    if $trans.master \notin MDev[j]$ then // Process master information
      $MDev[j] \leftarrow MDev[j] \bigcup trans.master$
    end if
    if $trans.fcode \notin MDev[j]$ then // Process function code
      $MDev[j] \leftarrow MDev[j] \bigcup trans.fcode$
    end if
    if $trans.fparameters \notin MDev[j]$ then // Process function parameters
      $MDev[j] \leftarrow MDev[j] \bigcup trans.fparameters$
    end if
  end while
```

memory, each operation and the corresponding memory contents are logged. These logs are useful for system monitoring, troubleshooting problems with devices, detecting system anomalies and reconstructing network events.

4. Experimental Results

A prototype implementation of the passive Modbus scanner was tested in a laboratory environment using two programmable logic controllers (PLCs), one master and one slave, in an Ethernet segment. Two experiments were conducted. The first experiment involved normal Modbus TCP communications between both devices. The second involved a rogue master. Modbus traffic was generated by having the master read two coils (associated with two PLC inputs) and then write a coil (associated with a PLC output) in a continuous loop.

The two input coils were located at memory addresses 3088 and 3152; the output coil was located at address 3104. The slave device was assigned the IP address `192.168.37.12` and was configured to respond to `192.168.37.11`, the IP address of a master.

In the first experiment, Modbus traffic was captured over a sufficiently long period of time to obtain most of the descriptive features of the network. The passive scanner then created a report based on the captured traffic (Figure 5a).

The report describing the network has four sections (for each slave device). The first section provides slave device ID information, i.e., the unit identifier, MAC address and IP address. The second section provides information about the master devices (IP and MAC addresses) that communicated with

```
||||||||||||||||||MODBUS DEVICE|||||||||||||||||||
 ---------------------Device ID----------------------
Unit Identifier:    255
Unit MAC address:   xx.xx.xx.20.49.31
Unit IP address:    192.168.37.12
 ---------------------Master ID----------------------
1.-master MAC:      xx.xx.xx.20.48.DE
1.-master IP:       192.168.37.11
 -------------------Function Codes-------------------
FC 15:    WRITE MULTIPLE COILS (OK)
FC 1:     READ COILS (OK)
 ---------Addressed Memories (MB references)---------
Address:  3088   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               ---Memory  Read---
Offset 0:      00000000  00000000
               ------------------
Address:  3088   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               ---Memory  Read---
Offset 0:      00000001  00000000
               ------------------
Address:  3104   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               --Memory Written--
Offset 0:      00000000  00000000
               ------------------
Address:  3104   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               --Memory Written--
Offset 0:      00000001  00000000
               ------------------
Address:  3104   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               --Memory Written--
Offset 0:      00000010  00000000
               ------------------
Address:  3152   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               ---Memory  Read---
Offset 0:      00000000  00000000
               ------------------
Address:  3152   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               ---Memory  Read---
Offset 0:      00000001  00000000
               ------------------
|||||||||-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-||||||||||
```

(a)

```
||||||||||||||||||MODBUS DEVICE|||||||||||||||||||
 ---------------------Device ID----------------------
Unit Identifier:    255
Unit MAC address:   xx.xx.xx.20.49.31
Unit IP address:    192.168.37.12
 ---------------------Master ID----------------------
1.-master MAC:      xx.xx.xx.20.48.DE
1.-master IP:       192.168.37.11
2.-master MAC:      xx.xx.xx.D6.1D.87
2.-master IP:       192.168.37.199
 -------------------Function Codes-------------------
FC 15:    WRITE MULTIPLE COILS (OK)
FC 1:     READ COILS (OK)
FC 43:    READ DEVICE IDENTIFICATION (Error) SubFC:14
 ---------Addressed Memories (MB references)---------
Address:  3088   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               ---Memory  Read---
Offset 0:      00000000  00000000
               ------------------
Address:  3104   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               --Memory Written--
Offset 0:      00000001  00000000
               ------------------
Address:  3104   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               --Memory Written--
Offset 0:      00000010  00000000
               ------------------
Address:  3152   Type:    COILS (read/write)
Master IP:  192.168.37.11   Status: accomplished
               ---Memory  Read---
Offset 0:      00000000  00000000
               ------------------
Type:    DIAGNOSTICS
Master IP:  192.168.37.199  Status: not-accomplished
               ------------------
|||||||||-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-||||||||||
```

(b)

Figure 5. Experimental results: (a) Normal operation; (b) Rogue master.

the slave device. The third section describes all the function codes associated with requests sent to the slave device and whether or not the associated replies indicated error conditions. The fourth section provides detailed information associated with memory access/update operations and diagnostics. For example, in Figure 5a, the report lists the values read from input coils at addresses 3088 and 3152, and the values written to output coil at address 3104.

The second experiment involved a rogue master unit, which was assigned the IP address `192.168.37.199`. Network traffic was captured for a shorter period of time than in the first experiment (resulting in a shorter report) but long enough to reveal the existence of the second master. The report in Figure 5b shows the presence of two masters. The report also shows that the rogue master attempted to execute function code 43 (encapsulated interface transport) with sub-code 14, a diagnostic function used to obtain (read) slave device identification information. Note that a negative response was obtained, i.e., the operation could not be completed.

5. Conclusions

The use of TCP/IP as a carrier protocol for SCADA systems and the interconnection of SCADA networks with IT networks opens pathways for remote access to industrial control systems. Security tools specifically designed for the SCADA networks are required because traditional IT security tools are generally not appropriate for industrial control systems. The passive scanner described in this paper provides valuable information about the state of Modbus networks, which are commonly used for pipeline operations in the oil and gas sector. The scanner is invaluable for tracking normal system operations as well as detecting anomalous events such as those caused by a rogue master device.

Acknowledgements

This work was partially supported by the Institute for Information Infrastructure Protection (I3P) under Award 2003-TK-TX-0003 from the Science and Technology Directorate of the U.S. Department of Homeland Security.

References

[1] American Petroleum Institute, API 1164: SCADA Security, Washington, DC, 2004.

[2] S. Boyer, *SCADA: Supervisory Control and Data Acquisition*, Instrumentation, Systems and Automation Society, Research Triangle Park, North Carolina, 2004.

[3] British Columbia Institute of Technology, Good Practice Guide on Firewall Deployment for SCADA and Process Control Networks, National Infrastructure Security Co-ordination Centre, London, United Kingdom, 2005.

[4] E. Byres, J. Carter, A. Elramly and D. Hoffman, Worlds in collision: Ethernet on the plant floor, *Proceedings of the ISA Emerging Technologies Conference*, 2002.

[5] Instrumentation Systems and Automation Society, Security Technologies for Manufacturing and Control Systems (ANSI/ISA-TR99.00.01-2004), Research Triangle Park, North Carolina, 2004.

[6] K. Mandia, C. Prosise and M. Pepe, *Incident Response and Computer Forensics*, McGraw-Hill/Osborne, Emeryville, California, 2003.

[7] Modbus IDA, MODBUS Application Protocol Specification v1.1a, North Grafton, Massachusetts (www.modbus.org/specs.php), June 4, 2004.

[8] Modbus IDA, MODBUS Messaging on TCP/IP Implementation Guide v1.0a, North Grafton, Massachusetts (www.modbus.org/specs.php), June 4, 2004.

[9] Modbus IDA, Modbus TCP is world leader in new ARC study, North Grafton, Massachusetts, (www.modbus.org/docs/Modbus_ARC_studyMay2005.pdf), 2005.

[10] Modbus.org, MODBUS over Serial Line Specification and Implementation Guide v1.0, North Grafton, Massachusetts (www. modbus.org/specs.php), February 12, 2002.

[11] Modicon, Inc., MODBUS Protocol Reference Guide, Document PI-MBUS-300 Rev. J, North Andover, Massachusetts, 1996.

[12] National Institute of Standards and Technology, System Protection Profile – Industrial Control Systems v1.0, Gaithersburg, Maryland, 2004.

[13] S. Northcutt and J. Novak, *Network Intrusion Detection: An Analyst's Handbook*, New Riders, Indianapolis, Indiana, 2002.

[14] S. Northcutt, L. Zeltser, S. Winters, K. Frederick and R. Ritchey, *Inside Network Perimeter Security: The Definitive Guide to Firewalls, VPNs, Routers and Intrusion Detection Systems*, New Riders, Indiana, 2002.

[15] M. Smith and M. Copps, DNP3 V3.00 Data Object Library Version 0.02, DNP Users Group, Pasadena, California, 1993.

[16] M. Smith and J. McFadyen, DNP V3.00 Data Link Layer Protocol Description, DNP Users Group, Pasadena, California, 2000.