Shiva Jyothi Angala

William Johnson

Pavan Madduri

A Proposed Framework for Detecting and Replaying Problematic Modbus TCP Traffic on a Physical or Simulated ICS Testbed

1. **INTRODUCTION**

This project serves as the beginning of a framework intended to allow for replay of malicious traffic on a simulated (and possibly physical) industrial control system testbed using Modbus TCP traffic. This framework will allow for the use of an ICS testbed as a honeypot which can visually replay detected attacks on the system. The completed system will consist of a network security monitor that can write to disk new packet capture data at the onset of an event that is most likely malicious, as well as a component that will replay that packet capture at a later date.

The current components that are implemented include the usage of the Bro network security monitor as well as a traffic replay engine that replays Modbus traffic from packet captures with approximately the same timing as the original traffic. If the environmental state of the system at the beginning of the original network traffic can be replicated (this may be difficult to do on a physical testbed as opposed to a simulation), replay of the traffic beginning with the initial malicious event should result in a visible replication of the same event.

1. **NETWORK SECURITY MONITORING**

Utilizing the Bro network security monitor [1], it is possible to monitor the state of a system at the application level, as well as modifying it as Modbus traffic is detected. Bro provides a rich scripting language that facilitates event handling (such as upon a read or write of a Modbus coil), data structures that can be maintained globally or locally, and functions to provide for easy modularization of behavior.

Bro monitors traffic from live or captured sources. As this project is primarily a network forensics project, for our purposes, we initially work with packet captures, but as the project progresses, once the system is joined, it will work with live traffic, creating packet capture files of indeterminate length as certain events are detected by Bro.

1. **BRO SYSTEM STATE MODEL**

We approach the task of monitoring our model system by utilizing the concept of critical state, mentioned in the works of Carcano et al. [2] and Fovino et al. [3]. A critical state is a state in which the system is unstable, creating a possibility of a failure of uncertain magnitude. For example, a natural gas pipeline in which the pressure is increasing to a point past the pipeline’s physical tolerances could be considered to be approaching a critical state.

By monitoring the state of the system, Bro can detect when a particular write to a component of the system would push that system into a critical state. In order to test Bro’s functionality in this regard, we built an example system that runs on an Allen-Bradley MicroLogix 1400b programmable logic controller.

For testing, the example system is a software-simulated water tank with three components. The water level in the tank is represented by an integer of the range 0-100 (the units are arbitrary). The tank has a pumping system to pump water into the tank, and there is a drainage valve that, when opened, drains the contents of the tank at approximately the same rate as the flow rate of the water from the pump. The pump and drain valve therefore can be represented with binary states.

Attached to the PLC itself is a trainer module that consists of four lights (blue, green, yellow, and red), two buttons (green and red), three switches, a voltage gauge, and an analog knob. For the purposes of our model, we only make use of the lights on the trainer. When the pump is on, its state is simulated with an illuminated blue light; when the pump is off, the light is not illuminated. The state of the drainage valve is represented by the green and red lights: when the valve is closed, the red light is illuminated, and when it is open, the green light is illuminated—the red or green lights should be lit exclusively, and one should always be on at any given time. In addition, the yellow light is illuminated if the tank is full (the water level is 100 units), and off otherwise.

For our purposes, we limit the water level at 100, and assume that the tank is an open tank, so if the pump is activated at that limit, the tank simply overflows. If the pump is on and the drain valve is closed, the water level is incremented by 1 unit per second. If the pump is off and the drain valve is open, the water level is decremented by 1 unit per second, until the water level is 0. The water flow logic is implemented simply by timers that reset once per second and continue or halt depending on the conditions at each tick.

In addition to the logic internal to the PLC, this system provides for Modbus access. The valve and pump conditions are stored as coils so they can be queried and modified by other devices on the network. The water level is stored in an input register, so it can be queried but not modified. These features allow for remote monitoring of the system state as well as control.

In the Bro scripting language, we represent the state of our system using a record data structure. Within that data structure, we represent the states of the pump and drainage valve as Boolean values, and the water level is represented by a count—an unsigned integer. While we initialize the values to preset initial states (the water level is 0, the pump is off, and the valve is closed), by updating the state of the model from events triggered by reads and writes to the coils and the input register, we can maintain an updated state fairly accurately. Regular state updates can be obtained by creating regular requests (such as one per second) by a Modbus TCP master. We provide for the status updates in our test system with a script using pymodbus to request coil and input register status updates once per second; this traffic is noticed by Bro and the model record is updated accordingly.

type Model: record {

water\_level: count;

valve\_closed: bool;

pump\_on: bool;

};

Fig. 1. Record data structure of our model for the Bro scripting engine

Through updates to this model caused by detected write events, we monitor (but do not drop) Modbus traffic throughout the system. There are some potential critical state events that can be immediately identified in our system. A primary example is activation of the pump while the drainage valve is sealed and the water is at its maximum level; this would cause an overflow of the tank. Another critical state event is activation of the pump while the drainage valve is open; water flowing into the tank will immediately continue through the drainage valve—this may not indicate a malicious event, but a technician may consider this event to be interesting and investigate further.

function check\_sanity() {

if (model$valve\_closed == F && model$pump\_on == T) {

print “ALERT: Pump enabled with unsealed drainage valve”;

}

if (model$pump\_on == T && model$water\_level >= 100) {

print “ALERT: Pump enabled with full container”;

}

}

Fig 2. A sample function used to check for state stability

event modbus\_write\_single\_coil\_response(c: connection, headers: ModbusHeaders, address: count, value: bool) {

if (address == 0) {

model$pump\_on = value;

}

if (address == 1) {

model$valve\_closed = value;

}

check\_sanity();

}

Fig 3. A sample Modbus event handler used to update the Bro model upon a coil write

Bro provides for a rich notification interface that can issue alerts in a number of ways (such as by email), but for our purposes, our alerts are simply strings printed to the screen. As the system enters into a critical state, we can see how this happened, and we can write rules for verbose logging as well.

For this model, our critical state notifications in Bro occur once a critical state is reached. For other models, Bro will be programmed in order to predict the approach of a critical state, such as the critical state prediction mentioned by Fovino et al [3].

1. **FORENSIC TRAFFIC INTERCEPTION AND REPLAY**

The second primary component of the project involves traffic replay for the forensic replication of significant events. As is, the system allows for Modbus traffic replay with timings approximate to the initial network communications.

Once traffic is initially recorded and saved as a .pcap file, it can be provided to the replay application which will automatically scan the packet capture for Modbus traffic, and replay this traffic to the network. If all of the destination IPs remain the same on the network, the replay software will

The replay application is implemented using Scapy [4] for packet parsing. Once a packet capture is provided, the application scans each packet at a time and determines whether it should be ignored or sent over the network. It first checks to see if the packet is a TCP/IP packet, as for this project, we are concerned primarily with TCP/IP traffic. Secondly, it checks to see whether the packet contains any data. If not (e.g., TCP handshake traffic), the packet is ignored.

If a packet has reached this point, it is checked to see if its destination port is 502. We limit the application to Modbus requests, but this restriction can be removed for more varied traffic—this may be desired in larger or more varied control networks. At this point, if this is the first packet that the system is interested in (first Modbus packet), the system first sleeps for 0.1 seconds to allow time for initial setup of the replay application; otherwise, the time of the current packet is checked. If there has not been a connection yet established with the destination IP, a TCP socket connection is established with the destination.

At this point, the time of the previous Modbus packet in the capture is subtracted from the time of the current Modbus packet for timing purposes. The application sleeps for the resulting time difference, and then proceeds. Here, the Modbus TCP payload of the current packet in the capture is extracted; and, as the system requires no source authentication, the payload can be sent through the appropriate socket. The recipient Modbus TCP slave device will accept the packet and respond accordingly.

This system has been tested both on a network with a single PLC running the water tank model, as well as the same network with an additional PLC. To create the initial traffic to be recorded, we programmed a script using pymodbus [5]. This script activates and deactivates the pump as well as drainage valve with varying speeds, in order to create recognizable access patterns that are visible on the PLC trainer lights. Our tests successfully recreated the recorded traffic patterns from packet captures at approximately the same speed as the initial script.

Ideally, packet captures that are replayed in the ICS system should begin with the first potentially malicious Modbus write. The capture must then proceed with all traffic in the capture to the desired ending point, as even writes from non-suspect sources contribute to the state of the system, and would therefore be required to be taken into account. As of now, the division of the packet capture must be done manually, using information from Bro’s alerts to determine the correct starting state and starting packet; automation of event-based packet capture can be done in future versions.

1. **FUTURE WORK**

There remain a number of potential additions that would improve the framework. First and foremost, if the Bro system is modified in order to automatically begin packet capture at the onset of a significant event, this would reduce the manual work required to prepare packet captures for system replay. In addition, the system could be modified in order for Bro to function on live traffic, creating packet capture files as needed rather than running on existing packet captures that must be modified to isolate the beginning of malicious or otherwise suspect network communications.

Additionally, the replay of Modbus traffic that causes the system to reach a critical state requires a replication of the system state. This is more plausible on a simulated testbed than a physical one. It could be somewhat possible on a physical testbed, but a primary issue lies with attacks that create intentional wear on parts of the system. While wear on a simulated testbed can be predicted, wear on a physical system is much less deterministic, and the initial health of the parts on a physical system can not be guaranteed to the same standards as the components of a simulated system. However, barring that, a valuable component to the overall forensic system would be a system that can replicate the original state of the testbed system as it reaches a critical state. Ideally, this would be something that could be controlled somewhat by the monitoring and replay framework, but much of the functionality here would need to be implemented within the testbed itself.

This system would ideally work in a number of situations that do not necessarily involve just Modbus, but more complex traffic. As is, the replay system can easily replicate Modbus traffic as there is no authentication, but in order to implement it for more secure or complicated protocols, more work would need to be done potentially at the application level as well as the transport and IP layers. For example, if an application layer protocol requires sequencing, in order to replay that traffic, care must be taken to modify the original packets to fit within an appropriate sequencing that the recipient devices would expect. Deep packet inspection therefore must be implemented for protocols such as that; the replay system would need to be better modularized in order to account for this as well.

While the timings in the replayed traffic seem appropriate at a glance, network and system overhead is not completely deterministic—there are unexpected delays and similar issues. While this can likely not be completely solved, depending on the system, the overhead of the Python interpreter used for the replay application may be unacceptable. As a proof of concept, it works well, but should be rewritten for better processing speeds, and would ideally also be run on a fairly dedicated system.

1. **CONCLUSION**

We presented the core and implemented portions of a framework that provides for critical state monitoring using Bro network security monitoring as well as replay of suspect traffic over a Modbus TCP ICS network. Bro allows for an observer to identify potentially malicious traffic through packet capture analysis using application-level state modeling. Secondarily, the replay application allows a forensic analyst to replay packet captures to review behavior of a system in response to that traffic once the state of the system has been matched with the state during the initial behavior; a functionality that should in the future be automated through communication of the system with the testbed.

1. **REFERENCES**

[1] The Bro Project. 2014. The Bro Network Security Monitor. Retrieved April 19, 2016 from <https://www.bro.org>.

[2] A. Carcano, A. Coletta, M. Guglielmi, M. Masera, I. Nai Fovino, and A. Trombetta. A Multidimensional Critical State Analysis for Detecting Intrusions in SCADA Systems. *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, May 2011.

[3] Igor Nai Fovino, Alessio Coletta, Andrea Carcano, and Marcelo Masera. Critical State-Based Filtering System for Securing SCADA Network Protocols. *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, October 2012.

[4] Scapy. Retrieved April 19, 2016 from <http://www.secdev.org/projects/scapy/>.

[5] Galen Collins. Pymodbus. Retrieved April 19, 2016 from <https://github.com/bashwork/pymodbus>.