DoS Exploitation of Allen-Bradley's Legacy Protocol through Fuzz Testing*

Francisco Tacliad
francisco.tacliad@gmail.com

Thuy D. Nguyen
Naval Postgraduate School
tdnguyen@nps.edu

Mark Gondree
Sonoma State University
gondree@sonoma.edu

ABSTRACT

EtherNet/IP is a TCP/IP-based industrial protocol commonly used in industrial control systems (ICS). TCP/IP connectivity to the outside world has enabled ICS operators to implement more agile practices, but it also has exposed these cyber-physical systems to cyber attacks. Using a custom Scapy-based fuzzer to test for implementation flaws in the EtherNet/IP software of commercial programmable logic controllers (PLC), we uncover a previously unreported denial-of-service (DoS) vulnerability in the Ethernet/IP implementation of the Rockwell Automation/Allen-Bradley MicroLogix 1100 PLC that, if exploited, can cause the PLC to fault. ICS-CERT recently announces this vulnerability in the security advisory ICSA-17-138-03. This paper describes this vulnerability, the development of an EtherNet/IP fuzzer, and an approach to remotely monitor for faults generated when fuzzing.

CCS CONCEPTS

•Security and privacy →Denial-of-service attacks;

KEYWORDS

Industrial control system, fuzz testing, EtherNet/IP, MicroLogix

ACM Reference format:
Francisco Tacliad, Thuy D. Nguyen, and Mark Gondree. 2017. DoS Exploitation of Allen-Bradley's Legacy Protocol through Fuzz Testing. In *Proceedings of Annual Industrial Control System Security Workshop, Orlando, Florida, USA, Dec. 2017 (ICSS'17),* 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Industrial control systems are vital components to the operation and functioning of Operational Technology (OT) systems used to manage critical infrastructure services. There are sixteen critical infrastructure sectors defined by the Department of Homeland Security (DHS) and most, if not all, utilize some form of ICS to manage and operate their assets [6]. OT systems are protected by varying levels of boundary defense, but often have exploitable network interiors. To identify cyber threats against the control network segment inside an OT system, vulnerabilities in the ICS protocols must be analyzed.

To serve the need for greater efficiency and automation, modern industrial network protocols have evolved from serial-based fieldbus protocols to TCP/IP-based protocols that are transported over standard Ethernet links. The Common Industrial Protocol (CIP) [26] and Ethernet/Industrial Protocol (EtherNet/IP) [27] are two well-known Open DeviceNet Vendors Association (ODVA) TCP/IP-based industrial protocols used by a large number of industrial automation vendors. Rockwell Automation/Allen-Bradley (RA/AB) PLCs (e.g., ControlLogix and MicroLogix) implement these protocols. Herein, unless explicitly specified, the term EtherNet/IP refers to both of these related protocols, collectively.

Fuzz testing, or fuzzing, is a penetration testing technique to verify the robustness of target software in handling invalid, malformed, or unexpected input data. Fuzzing the implementations of control network protocols is an important step towards developing more secure industrial control systems. Voyiatzis *et al.* argue that control networks are rich targets for this type of black-box testing because those systems are likely to have been developed years ago, the source code and specification may not be available, a variety of vendor-specific implementations may exist, and Internet connectivity is increasingly integrated with such systems [36].

Little information has been made publicly available on the vulnerabilities of the EtherNet/IP software used in commercial PLCs. To examine the robustness of the EtherNet/IP implemenation of select RA/AB devices, we create a fuzz testing tool (ENIP Fuzz) using Scapy, a Python module used for packet parsing and crafting [1]. Scapy's flexibility to send, sniff, dissect and forge network packets has made it a popular tool among penetration testers.

Using ENIP Fuzz, we discover a previously unreported vulnerability in the EtherNet/IP implementation of the Rockwell Automation MicroLogix 1100 PLC that, if exploited, can cause the MicroLogix PLC to become unresponsive. The ICS-CERT security advisory ICSA-17-138-03 [3] identifies several critical infrastructure sectors that are potentially vulnerable to this network denial-of-service attack, i.e., Critical Manufacturing, Food and Agriculture, Transportation Systems, and Water and Wastewater Systems.

In summary, this paper describes the following contributions:

(1) A Scapy-based fuzzer for exploiting EtherNet/IP security vulnerabilities.
(2) A remote fault detection strategy employing a liveness check, unexpected responses, and performance measurement to monitor the remote device during testing.
(3) A discovery of a deficiency in MicroLogix's handling of the Programmable Controller Communication Commands (PCCC) protocol [14], which is transported inside CIP messages. PCCC

*The views expressed in this material are those of the authors and do not reflect the official policy or position of the Naval Postgraduate School, the Department of Defense, or the U.S. Government.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSS'17, Orlando, Florida, USA
© 2017 ACM. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

is a vendor-specific CIP extension, used for communications with legacy RA/AB PLCs. By sending a specially crafted PCCC command, a remote, unauthenticated attacker can trigger an unrecoverable error condition, requiring the PLC to undergo a hard reset.

(4) A preliminary exploration of potential cross-generational vulnerabilities in different families of RA/AB PLCs.

In the remaining sections, we provide basic information on EtherNet/IP protocols and review prior work in §2. We then describe ENIP Fuzz and our remote monitoring approach in §3. A discussion of the fault detection results, the MicroLogix vulnerability, and the ControlLogix experimentation is in §4. We conclude in §5.

2 BACKGROUND AND RELATED WORK

This section provides an overview of the three protocols relevant to this work and summarizes previous work in ICS fuzz testing.

2.1 EtherNet/IP Protocols

CIP. Being an object-oriented protocol, each node in a CIP network is modeled as a set of *objects* [26]. An object is an abstract representation of a particular component within a product. A *class* is a set of objects of the same kind of system component. An *object instance* is the actual representation of a particular object. An *instance* of a class or an object share the same attributes, but has its own unique attribute values [26]. For example, the Identity object identifies the device, and its Status attribute (attribute ID = 0x05) describes the current state of the entire device [26]. A CIP node can have multiple object instances within a class of objects. A group of objects used in a device is referred to as that device's *object model* [26]. The CIP object library supports network communications, network services, and automation functions used by industrial components such as analog and digital input/output devices.

EtherNet/IP. This protocol is an adaption of CIP to allow CIP communications to be transported over standard Ethernet. The EtherNet/IP standard defines port 44818 as the designated port over which EtherNet/IP devices accept TCP and UDP connections. EtherNet/IP supports two primary types of communications: implicit and explicit [27].

Implicit messaging enables a sending device (i.e., the producer) to exchange scheduled, time-critical control data to one or more receiving devices (i.e., the consumers) [27]. With implicit messaging, a CIP connection must be established [27]. Communication sessions related to a specific connection are assigned a unique connection identifier upon establishing a connection [27]. The CIP connection identifier acts as a dedicated communication path allowing multiple end-points to share data without the need to send the data multiple times [27]. Implicit messaging uses UDP and can be unicast or multicast [27].

Explicit messaging provides general request/reply communication between two devices and is used for non-real-time data. For EtherNet/IP, explicit messaging uses TCP and does not require establishing a CIP connection [27].

PCCC. This protocol provides legacy support for older RA/AB PLCs, e.g., PLC5 and SLC500 [17]. When used with EtherNet/IP, the PCCC object (class code = 0x67) processes PCCC messages encapsulated in CIP payloads. This encapsulation is accomplished through the use of the "Execute_PCCC" CIP service (service code = 0x4B). Each PCCC message contains a command code and a function code, which together specify the PCCC command to be executed by the receiving device. For example, the command code 0x06 and the function code 0x00 indicate the *echo* command whereas the command code 0x0F and the function code 0xA2 specify the *protected typed logical read with three address fields* command.

2.2 Fuzzing Methodologies

While there is no universally-accepted taxonomy of fuzzing approaches, most of the literature places fuzzers into one of two categories: *mutation-based* and *generation-based*. Mutation-based fuzzers apply transformations (mutations) on existing data samples to create test cases [30]. Generation-based fuzzers create test cases from behavior models of the system under test (SUT). Each method has its own strengths and weaknesses.

Mutation-based Fuzzers. Mutation-based fuzzers modify valid inputs by altering bytes to create fuzzed inputs [30]. Some mutation fuzzers utilize a description of the input fields, while other mutation fuzzers do not require any knowledge of the format; instead, they use heuristics to guess field structure and mutate each field [30]. Most mutation fuzzers extract data from recorded sessions for mutation, although some fuzzers intercept and mutate live traffic [30]. Mutation-based fuzzing is considered a form of brute force testing in that the fuzzer starts with valid inputs and incrementally transforms every bit within the input [33]. This requires little up-front research and implementing a naive mutation-based is relatively straightforward. The SUT may employ complex logic infrequently invoked. Many fuzzing iterations may be required to achieve sufficient code coverage, though this challenge can be offset with automation.

Generation-based Fuzzers. Generation-based fuzzers construct test cases employing rules defining a grammar-based specification for inputs. The simplest fuzzers of this type create input data of random strings of bytes [30]. Some generation-based fuzzers must be configured using an input description or data model to generate test cases [33]. The generation-based approach requires up-front research to understand the specification or source code of the target. However, rather than using hard-coded test cases, a generation-based fuzzer uses grammar-based rules to dynamically pinpoint the portions of the file or packet that represent fuzzable variables.

2.3 ICS Protocol Fuzzers

We survey relevant fuzzers and fuzzing frameworks, highlighting, when applicable, those ICS protocols each supports (Table 1). We classify the surveyed software as either a *custom fuzzer* or a *fuzzing framework*. Custom, or one-off, fuzzers target a specific file format or network protocol. They can be used to stress test a wide range of applications that support the target format or protocol.

Sutton [33] descibes fuzzing frameworks as homogenous development environments that enable the use of reusable utilities to maximize extensiblity. Sulley and Peach are examples of open-source, generation-based fuzzing frameworks that support some ICS protocols [30]. Sulley is a framework consisting of multiple

Table 1: Summary of ICS Fuzzers

Name	Type	Protocol	Availability
Aegis Fuzzer [10]	custom	DNP3, Modbus	commercially licensed, early version open-source
Beyond Security's beSTORM [5]	framework	several, including DNP3	commercially licensed
blackPeer [19]	framework	several, including Modbus	NA
Codenomicon's Defensics [7]	framework	several, including CIP, EtherNet/IP, Modbus, OPC UA Server, Profinet, Scada GOOSE	commercially licensed
ICCP Fuzzer [22]	custom	ICCP	NA
LZFuzz [18]	framework	several, including SNMP [30]	NA
MTF [36]	custom	Modbus	NA
OPC-MFuzzer [37]	custom	OPC, DCOM, RPC [29]	NA
OPC Server Fuzzer [24]	custom	OPC Server	NA
Peach [9]	framework	several, including Modbus, BACNet, DNP3, OPC [9, 37]	open-source
ProFuzz [23]	custom	Profinet	open-source
scada-tools [34, 35]	custom	Profinet	open-source
Sulley [28]	framework	several, including Modbus, DNP3, TPKT, COPT [20]	open-source
Wuldtech's Achilles [4]	custom	several, including EtherNet/IP, Foundation Fieldbus, MMS, Modbus, OPC UA, Profinet, DNP3, MMS, SES-92	commercially licensed

extensible components, including an instrument to monitor the health status of the target and detect, track, and categorize what sequence of test cases triggers faults [33]. Sulley can also fuzz in parallel, increasing performance [33].

While some commercial fuzzers report supporting EtherNetIP in some fashion ([7], [4]), no other surveyed fuzzers support EtherNet/IP at all. Smith and Francia [31] report on an EtherNet/IP and CIP fuzzer, but the code is not available. The Modbus/TCP Fuzzer (MTF) and scada-tools are two custom Scapy-based fuzzers for Modbus and Profinet, respectively [34, 36]. At DEFCON 15, Devarajan describes using the Sulley framework to fuzz Modbus, DNP3 and ICCP [20]. Similarly, Peach is designed for flexiblity. It provides custom fuzzing strategies and data modifiers, as well as special processes called Agents for fault detection [9].

3 DESIGN AND IMPLEMENTATION

In general, fuzzers operate under two basic assumptions: (i) faults contained in a target application can be triggered through input controlled by the user and (ii) the execution of a faulty portion of an application will result in some behavioral manifestation (e.g., bricking the device or producing unexpected output) [12]. Most systems are designed to work with specific inputs but, ideally, should be robust enough to gracefully handle malformed data. Therefore, flaws found from fuzzing will correspond to a bug in the target (e.g., file, network protocol, embedded device, and software).

3.1 Implementation of Support Library

We implement ENIP Fuzz, a custom fuzzer for testing security vulnerabilities in the EtherNet/IP and CIP layers of an EtherNet/IP implementation. ENIP Fuzz implements its own EtherNet/IP support library using Scapy, a Python module used for packet crafting and manipulation [1]. Our library conforms to the EtherNet/IP specifications [26, 27] and is based on an existing Wireshark dissector for EtherNet/IP [38], written in C. ENIP Fuzz is not a one-to-one translation of Wireshark's source code. Internally, they are not the same; instead, Wireshark is used for validating data field structure rather than reuse of its parsing logic. In fact, we discover errors in Wireshark's implementation of the CIP Common Services, specifically the Multiple Service Packet [26, §A-4.10]. Additionally, Wireshark does not support proprietary vendor-specific EtherNet/IP implementations, such as the PCCC protocol [25].

ENIP Fuzz is complete in its support of the EtherNet/IP specification [27] and approximately one fourth of the CIP specification [26]. To characterize the EtherNet/IP traffic space we collect several samples of communication from our ICS lab environment, which included the AB/RA MicroLogix 1100 and ControlLogix 5570 devices. We implement all EtherNet/IP and CIP services captured in these traffic samples and add support for PCCC [25].

3.2 Implementation of Fuzzer

Based on observed traffic in our lab, three types of EtherNet/IP service requests were chosen as test cases to fuzz: EtherNet/IP Register Session, CIP NOP, and Execute PCCC Service. For each type, fields are selected as primitives to fuzz based on the observed volatility in the field's value. Fields that remained static after having been assigned a constant value (e.g., a field used for identifying an established EtherNet/IP session) are not fuzzed. Additionally, fuzzing is performed only at the layer in which the service request is encapsulated.

EtherNet/IP Register Session Request. The EtherNet/IP Register Session request is used for establishing a TCP encapsulation session between an originator and a target. As defined by the specficiation [27, §2-4.4], the originator shall open a TCP/IP connection to the target on port 0xAF12; the originator shall then send an EtherNet/IP Register Session request to the target. Upon receiving a valid Register Session request, the target shall assign and reply with a unique session identifier called the Session Handle, an unsigned 32-bit integer value [27].

To fuzz the EtherNet/IP Register Session request, we manipulate the Protocol Version and Options Flags, first in isolation and then simultaneously. Both fields take the value of an unsigned 16-bit integer. For each test case, ENIP Fuzz is programmed to fuzz these fields with a random integer from 0 to 65535. Per the EtherNet/IP specification [27, §2-4.4] and experimentation, with the exception of Session Handle, these are the only non-constant fields.

CIP NOP Request. The CIP NOP (No Operation) request is a CIP common service that causes the receiver to generate a No Operation response [26, §A-4.17]. The receiver does not execute any other internal action; if the receiver does not support the CIP NOP, a response with a status error is returned [26, §A-4.17]. The CIP NOP request is chosen because of its simplicity. The CIP NOP request has no specified data field structure and is only embedded in an EtherNet/IP Send RR Data Packet.

Without any associated data field, the Class and Instance fields within the Request Path are the only fuzzable variables for CIP NOP. They are fuzzed individually and then at the same time. Class and Instance are a type of CIP segment used for referencing a specific CIP entity [26]. Segments are grouped together in order to define a relationship among different objects. The Request Path is a value used to specify such a relationship.

Execute PCCC Service. PCCC is a vendor-specific application-layer protocol used for communication between certain RA/AB processors [25]. Unlike the EtherNet/IP Register Session and CIP NOP, the Execute PCCC Service is not a common service. According to its specification, PCCC is used primarily to "ease communication between legacy networks and the new CIP networks" [14, p. 7.17]. EtherNet/IP products are able to support PCCC through encapsulation within CIP. In our lab, we observe that the RSLogix 500 software, used to program ladder logic for RA/AB PLCs, periodically sends Execute PCCC Service commands to the PLC. The high regularity with which RSLogix sends the Execute PCCC Service command is the motivating factor in its selection for fuzzing.

The Protected Typed Logical Read with Three Address Fields command is the specific Execute PCCC Service function chosen for fuzzing. This function is used to read data from a logical address [14, p. 7.17]. To fuzz this function the following fields are manipulated in isolation and then in combination: Byte Size, File Number, File Type, and Element Number.

3.3 Fault Monitoring

Though fuzzers may differ in their fuzzing techniques, fault monitoring is of particular importance. At its most basic level, a fuzzer might detect that a fault was triggered if the target crashes or becomes bricked, i.e., application is rendered unusable or is unable to accept a new connection [33]. More sophisticated fault detection may be achieved with the help of a debugger. For example, the Peach and Sulley fuzzing frameworks communicate directly with a debugger attached to the target application [9, 33]. Sutton *et al.* propose an alternative, where a debugger runs on the target platform to monitor exceptions and correlate fuzzing behavior with observed faults [33].

There are three ways the fuzzer remotely monitors for faults generated when fuzzing: a liveness check, unexpected responses, and performance degradation.

Many existing fuzzing approaches attach a debugger to the SUT to determine when crashes occur. For example, Basnight uses an available JTAG interface for debugging the RA/AB ControlLogix L61 CPU [13]. Debugging with JTAG requires special pins called test access ports which may not be available in all devices. Other studies have leveraged built-in fault monitoring utilities. Dunlap describes using a task monitor utility available in the ControlLogix L61 to access timing data from ladder logic execution times for an anomaly-based intrusion detection system [21]. Attaching a debugger or performance monitor is not an option for our experiments, so we adopt alternative, remote-fault monitoring methods.

Since explicit interaces for fault detection are not always available, people have used remote analysis to determine when crashes have occured. Shapiro *et al.* describe using a *liveness check* to identify when an ICS device revives itself during a fuzzing session [30]. Their study suggests that for protocols running over TCP, the occurrence of a TCP RST flag is a sufficient metric for indicating that a target device has crashed; however they concede that this method may produce false positives. Similarly, Voyiatzis *et al.* argue that direct access to the SUT is not needed, simply a network connection to it [36]. They suggest that through network behavior—such as socket timeout, reset, or close; failure in reopening a closed socket; and failure in opening a new socket—are useful indications that the SUT has crashed. ENIP Fuzz utilizes such indicators to judge if the target has crashed.

ENIP Fuzz also filters for unexpected responses. Voyiatzis *et al.* record information during fuzzing the Modbus protocol to check if responses are outside of the specification [36]. Similarly, ENIP Fuzz inspects response packet data for responses that do not conform to the specification.

In addition, we consider performance degradation as fault for real-time systems that may be elicited during fuzz testing. To the best of our knowledge, no other study has considered performance as a type of fault for detection during fuzzing. The NIST Guide to Supervisory Control and Data (SCADA) and Industrial Control Systems Security highlights that ICSs are generally *time critical*; where delay of information cannot be tolerated [32]. Thus, malformed packets impacting the timely delivery of responses may be considered a type of soft failure, causing the SUT to go outside normal behavior. One of the contributions of our study is in exploring three potential performance metrics during fuzzing (discussed further in Section 4.3) to ascertain their reasonableness as candidates for detecting these types of soft failures.

Generally, for monitoring performance when fuzzing, we record a baseline of valid traffic for each generator and compare this to traffic captured during fuzzing. These baselines serve as a control, modeling how the device should behave under normal operation (e.g., valid EtherNet/IP requests). Records captured during fuzzing are compared to the baseline and analyzed for irregularities in response times. Any anomalous behavior is correlated with fuzz scenario, using timestamps and packet inspection.

3.4 Test Environment

Our test environment consists of four components: the SUT, the fuzzer, background traffic generators, and the monitor (Fig 1). The test equipment for the experiments consists of an Allen-Bradley

MicroLogix 1100 PLC, a Windows 7 Virtual Machine (VM) with RSLinx, a Kali 2.0 VM with the fuzzer, a Kali 2.0 VM with the Ping utility, and a workstation with Mac OS X running Wireshark. The equipment are connected via Ethernet to a common hub. The SUT employed in this study is the MicroLogix 1100 PLC. The MicroLogix 1100 is an EtherNet/IP I/O scanner device that supports explicit messaging. Experimental traffic sent to the SUT is generated by a Kali 2.0 host running ENIP Fuzz. The background traffic generators are two hosts: Kali 2.0 running Ping and Kali 2.0 running RSLinx. The Ping utility is used to send ICMP Echo requests at one second intervals. RSLinx is software for Allen-Bradley devices used to browse and configure PLC devices. To generate requests, RSLinx is set to "autobrowse" mode, causing it to send UDP broadcast EtherNet/IP List Identity Response requests to the SUT (and, in fact all devices connected to the network). The monitor is Mac OS X host running Wireshark to collect all traffic for analysis.

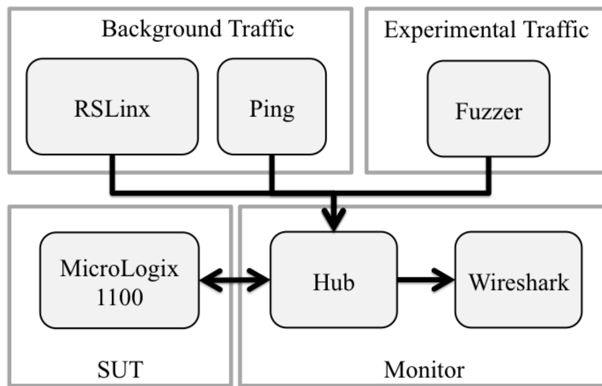

Figure 1: Fuzzing test environment

During experimentation, a liveness check is performed using the Ping utility to determine that the target is still responsive. For performance degradation, we monitor the latency in responses to both ICMP Echo requests and EtherNet/IP requests made by the RSLinx. Irregularities in recorded response times may suggest increased CPU utilization or memory exhaustion related to fuzz testing. The SUT is also monitored for unexpected responses, i.e., responses outside the EtherNet/IP specification or otherwise incorrect (e.g., responses that contain erroneous data).

4 RESULTS AND ANALYSIS

We use three metrics for analysis: the deltas between ICMP Echo requests from Ping, List Identity requests from RSLinx, and the response from the service request being fuzzed. The SUT interacts with the traffic generators for about 5 minutes during a "warm-up period," after which the fuzzer sends either correctly formed packets (during baseline) or malformed packets (during testing) for a period of approximately 20 minutes. The Wireshark packet capture of the fuzzing session is then truncated into a 10 minute window, after which each of the metrics is analyzed. Each delta is calculated by taking the difference between the timestamp of the response and the request.

We perform three baseline measurements and fourteen trials (Table 2). Each baseline and trial is repeated twice making the total number of tests twenty-eight.

Table 2: ICS Fuzzing Trials

Trial Name	Field Fuzzed
enip-register-session-baseline	NA
enip-register-session-fuzz-protocol-version	Version
enip-register-session-fuzz-option-flag	Options
enip-register-session-fuzz-protocol-option	Version,Options
cip-nop-baseline	NA
cip-nop-fuzz-class	Class
cip-nop-fuzz-instance	Instance
cip-nop-fuzz-class-instance	Class,Instance
pccc-exec-baseline	NA
pccc-exec-fuzz-byte	Byte Size
pccc-exec-fuzz-file-no	File Number
pccc-exec-fuzz-file-type	File Type
pccc-exec-fuzz-element	Element No.
pccc-exec-fuzz-all	File No., File Type, Element No.

4.1 Response Time Analysis

Our results suggest that using the deltas in response times from ICMP Echo requests and List Identity requests may not be meaningful metrics for determining whether fuzzing has an observable effect on the performance of the SUT. Using Tukey's Honest Significant Difference (HSD) test there is no significant difference in response times when fuzzing compared to when sending non-malformed traffic. Fig 2 and Fig 3 illustrate Tukey's HSD graphs for the fuzzing metric with the EtherNet/IP Register Session and CIP NOP commands, respectively. In this example, we see that all populations appear to overlap, therefore the null hypothesis (that the samples represent the same distribution, i.e., the latencies are unaffected) cannot be rejected.

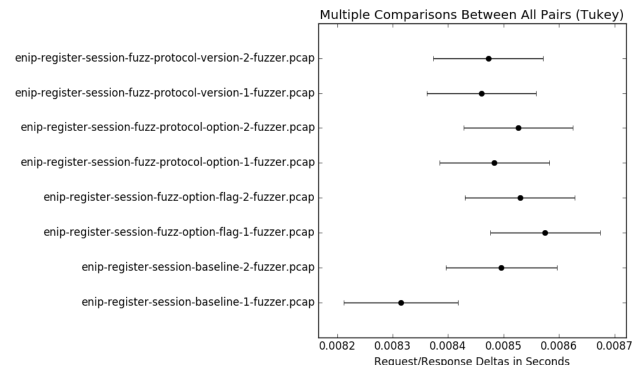

Figure 2: Tukey's HSD for Register Session Tests

On the other hand, under Tukey's HSD for tests against the Execute PCCC Service command, we observe some sensitivity with

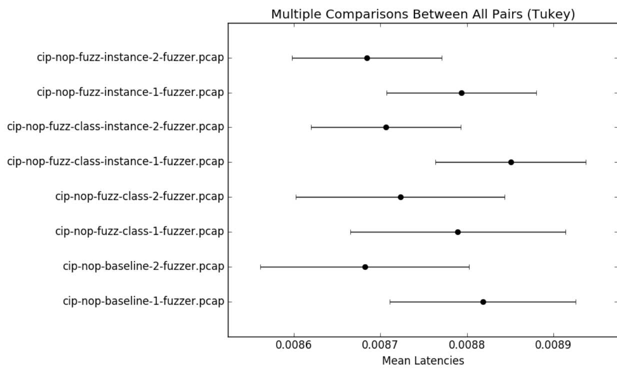

Figure 3: Tukey's HSD for CIP NOP Tests

these performance metrics (see Fig 4). Tukey's HSD suggests performance may be impacted during analysis, however the results are inconsistent. For example, when comparing pccc-exec-fuzz-file-no-1 and pccc-exec-fuzz-file-no-2 we expect the mean latencies to overlap based on tests performed on EtherNet/IP Register Session and CIP NOP test cases, but instead we observe a statistical difference between these populations. We observe similar anomalous results when comparing pccc-exec-fuzz-byte-1 with pccc-exec-fuzz-byte-2. Since there is high variability in the traffic contents across fuzzing sessions, the fact that we may see variable behavior across sessions is not unexpected; but more testing is warranted before it is possible to claim fuzzed inputs were responsible for any apparent performance degradation.

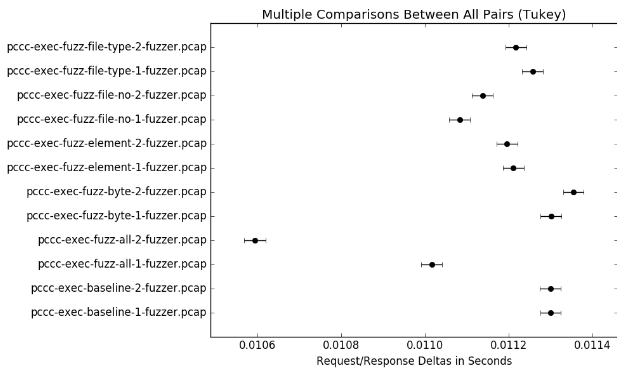

Figure 4: Tukey's HSD Tests of Execute PCCC Service

4.2 Denial-of-Service Fault

When fuzzing the Execute PCCC Service, we discover a previously unreported DoS vulnerability caused by accessing certain Data Files with an invalid File Type. This result is not represented in the deltas discussed previously; we identify the types of packets that cause the fault and bypass it to produce the results in Fig 4. By sending a specially crafted Execute PCCC Service packet to the SUT, a Major Error (0x08) is triggered and the device becomes unresponsive. To clear the fault, the device must be power-cycled and reset using the RSLogix Clear Major Fault utility. The SUT used to test the fault condition is a MicroLogix 1100 PLC (1763-L16BWA Series B, FRN 14).

According to the MicroLogix 1100 reference manual, data files store status and data information associated with instructions used in ladder subroutines [15, p. 40–41]. An existing CVE (CVE-2012-4690) describes a DoS fault that can occur when a malformed CIP packet is written to the Status file [2]. It is not clear if these two faults are related. Allen-Bradley has issued firmware releases for the MicroLogix 1100 to mitigate that vulnerability; the anomaly identified in CVE-2012-4690 was corrected in FRN 13 according to the release notes for FRN 14: "Status file bits [...] were writable through communication messages which allowed the possibility to force the controller to go into fault. The solution included in this firmware revision allows users to CLEAR these bits [...] but does NOT allow them to SET using Communication Messages" [16, p. 5]. Moreover, the observed fault is generated by a read request, i.e., eliciting our fault does not involve any write requests. Fig 5 shows the output of the MicroLogix 1100 Status File while the SUT is in the Faulted state.

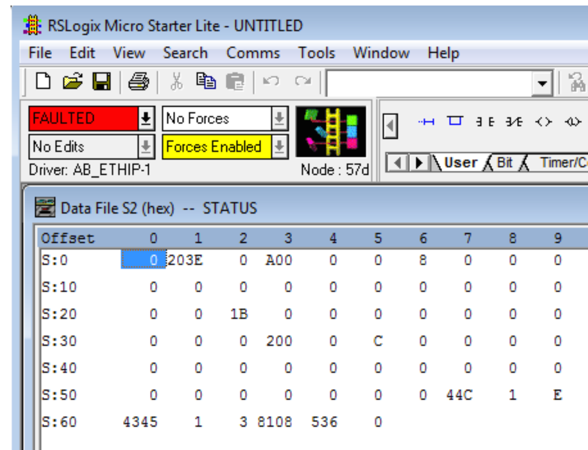

Figure 5: MicroLogix 1100 Status File

To exploit the vulnerability, the attacker sends a single Execute PCCC Service - Protected Typed Logical Read with Three Address Fields packet with a File Number of 0x02–0x08 and File Type 0x48 or 0x47. Any combination of File Number 0x02–0x08 and File Type 0x48 or 0x47 will trigger a Major Error (0x08). Figures 6 and 7 illustrate example packets that will cause the fault.

In addition, to reproduce the fault, it appears that the attacker must establish a session with the target with an EtherNet/IP Register Session Request and then create a connection instance with a Connection Manager Forward Open Request. The DoS packet needs not to immediately follow the Connection Manager Forward Open Request to cause the fault.

4.3 ControlLogix Experiment

We speculate that the same PCCC vulnerability could potentially exist in newer RA/AB PLC models because legacy code tends to be left in newer software without being fully tested. However,

```
###[ ENIP TCP ]###
          Command   = Send Unit Data (0x0070)
          Length    = 45
          Session_Handle= 0x6077596d
          Status    = Success
          Sender_Context= 0
          Options   = 0
###[ Send Unit Data ]###
             Interface_Handle= 0
             Timeout   = 20
###[ ENIP_CommonPacketFormat ]###
                Item_Count= 2
                \Items     \
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connection-Based (0x00A1)
                 |  Address_Length= 4
                 |  Connection_Identifier= 0x6d596902
                 |###[ Common Packet Format Item ]###
                 |  Address_Data_Item= Connected Transport Packet (0x00B1)
                 |  Data_Length= 25
                 |  Sequence_Number= 0x1
###[ Common_Industrial_Protocol ]###
                   Request_Response= Request
                   Common_Service= Execute_PCCC_Service
                   Request_Path_Size= 2
                   \Words     \
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Class ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Class     = 0x67
                    |###[ CIP Request Path ]###
                    |  Path_Segment_Type= Logical Segment
                    |  Logical_Segment_Type= Instance ID
                    |  Logical_Segment_Format= 8-bit logical address
                    |  Eight_bit_Instance= 0x1
###[ CIP Execute PCCC Service Request ]###
                     Length_of_Requestor_ID= 7
                     CIP_Vendor_ID_of_Requestor= Rockwell Software, Inc.
                     CIP_Serial_Number= 90180339
                     CMD       = 0x0F
                     Status    = 0x0
                     Transaction_Word= 2
                     Function  = Protected Typed Logical Read Three Address Fields
                     Byte_Size = 0x0
                     File_No   = 0x5
                     File_Type = 0x47
                     Element_No= 0x0
                     Sub_Element_No= 0x0
```

Figure 6: A DoS Packet, highlighting fields encapsulated at the PCCC layer.

```
0000   00 1d 9c a1 28 4c 08 00 27 ec 11 8c 08 00 45 00
0010   00 6d dc c9 40 00 40 06 db f7 c0 a8 00 3e c0 a8
0020   00 3b bb 5c af 12 60 0e 60 ff 4d 53 0d 78 50 18
0030   72 10 e3 0f 00 00 70 00 2d 00 6d 59 77 60 00 00
0040   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050   00 00 14 00 02 00 a1 00 04 00 02 69 59 6d b1 00
0060   19 00 01 00 4b 02 20 67 24 01 07 4d 00 f3 0a 60
0070   05 0f 00 02 00 a2 00 05 47 00 00
```

Figure 7: Packet Structure of a DoS Packet, highlighting fields encapsulated at the PCCC layer.

running the same PCCC stress tests on the ControlLogix 5570 did not cause the expected DoS fault condition. Nevertheless, this experiment yields insight into the differences in the way MicroLogix and ControlLogix respond to the Protected Typed Logical Read with Three Address Fields PCCC command.

A PCCC reply always has a *status (STS)* byte, and for some commands, an *extended status (EXT STS)* byte. Fig 8 shows the packet format of the Protected Typed Logical Read with Three Address Fields [14].

C	DST	SRC	CMD 0F	STS	TNS	FNC A2	Byte Size	File No.	File Type	Ele. No.	S/Ele. No.
R	SRC	DST	CMD 4F	STS	TNS	DATA				EXT STS	

Figure 8: Packet Format of Protected Typed Logical Read with Three Address Fields Command [14]

We observe that MicroLogix only returns the STS byte (0x10 = "Illegal command or format") whereas ControlLogix returns both STS and EXT STS bytes—STS = 0xF0 ("Error code in the EXT STS byte") and EXT STS = 0x06 ("Address doesn't point to something usable") [14]. This functional difference indicates that it may be more valuable to fingerprint PLCs using information at the application-level protocol headers, rather than more generic techniques using just port numbers.

5 CONCLUSION AND FUTURE WORK

In this paper, we describe ENIP Fuzz, a fuzzing tool developed to uncover vulnerabilities in the EtherNet/IP software used in commercial PLCs. ENIP Fuzz can dissect Ethernet/IP packets with encapsulated CIP and PCCC messages. We use two different RA/AB PLCs, i.e., MicroLogix 1100 and ControlLogix 5570 as the SUT for our fuzzer.

While stress testing the MicroLogix 1100 PLC's handling of PCCC messages, we discover a flaw in its implementation of the Execute PCCC Service request. Successful exploitation of this vulnerability by sending a single, specially-crafted PCCC packet could cause the PLC to enter a faulted state that must be power-cycled and reset using a special recovery tool. This improper input validation vulnerability has been confirmed by the vendor and documented in the ICS-CERT security advisory ICSA-17-138-03. As reported by the advisory, the attack affects all existing models of the MicroLogix 1100 family. Testing beyond this family, we repeat the same fuzz testing on the ControlLogix 5570 and observe no failure.

Another contribution is the use of response times as a metric for remote fault detection. For the data we collect, statistical hypothesis testing via Tukey's HSD suggests we observed no significant difference between the response times during normal activity and during fuzz testing; however, we encourage the community to consider metrics like these and consider performance degradation as a fault condition for real-time systems. Developing more nuanced remote fault-detection metrics for fuzz testing (rather than the current crash/no-crash metrics) seems well-intentioned but non-trivial.

5.1 Future Work

As an extension to this work, the EtherNet/IP support library can be expanded so that it is fully compliant with the EtherNet/IP specifications. Better handling of proprietary protocols such as PCCC should also be added; currently, these protocols are not supported by Wireshark's dissectors, and thus must be validated through alternative means, such as manual inspection of traffic (or, in our case, custom tools like ENIP Fuzz). We plan to integrate ENIP Fuzz into the Metasploit framework, either as a new module or an extension to an existing module.

We had initially considered those protocol layers targeted by existing fuzzing tools as out-of-scope of our work; however, testing the TCP and IP layers of the network stack may expose vulnerabilities in the ENIP/IP implementation, as the specification makes certain assumptions about the underlying TCP/IP mechanisms.

We consider the investigation of related flaws across products—i.e., derived from specification ambiguities or from the irregularities of handling reserved or rare legacy protocols—to be potentially very interesting. Such shared flaws have been observed in many other products, but no comprehensive study exists for families of ICS

devices. Thus, it may be fruitful to further explore EtherNet/IP implementations across related products, i.e., products that conforms to the ODVA specification or deemed interoperable with related models. In particular, OpENer is a POSIX-compliant implementation of an EtherNet/IP protocol stack [8]. The development of this stack is partially supported by Rockwell Automation [11]. Given its portability, OpENer would be quite amenable to testing using existing frameworks and black-box fuzzers.

ACKNOWLEDGMENTS

We would like to thank LCDR James Gormley for his assistance in conducting the ControlLogix experiments, and DHS ICS-CERT for their assistance in responsible disclosure with the affected vendor.

REFERENCES

[1] 2011. Scapy. (2011). http://www.secdev.org/projects/scapy Accessed: Aug. 14, 2016.
[2] 2012. CVE-2012-4690. (2012). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4690 Accessed Aug. 14, 2016.
[3] 2017. Advisory ICSA-17-138-03. (2017). https://ics-cert.us-cert.gov/advisories/ICSA-17-138-03 Accessed Sept. 3, 2017.
[4] n.d.. Achilles Platform. (n.d.). https://www.wurldtech.com/product/achilles Accessed: March 25, 2016.
[5] n.d.. BeSTORM software software security testing tool. (n.d.). http://www.beyondsecurity.com/bestorm.html Accessed: March 25, 2016.
[6] n.d.. Critical Infrastructure Sectors. (n.d.). https://www.dhs.gov/critical-infrastructure-sectors Accessed Aug. 14, 2016.
[7] n.d.. Defensics. (n.d.). http://www.codenomicon.com/products/defensics/ Accessed: March 25, 2016.
[8] n.d.. OpENer. (n.d.). https://github.com/EIPStackGroup/OpENer Accessed: Sept. 8, 2017.
[9] n.d.. Peach Introduction. (n.d.). http://community.peachfuzzer.com/Introduction.html Accessed: March 25, 2016.
[10] Automatak. [n. d.]. ([n. d.]). https://github.com/ITI/ICS-Security-Tools/tree/master/protocols Accessed March 25, 2016.
[11] Rockwell Automation. 2009. Rockwell Automation Sponsors Development of Open-Source Software Stack. (2009). http://phx.corporate-ir.net/phoenix.zhtml?c=196186&p=irol-newsArticle&ID=1356918 Accessed Sept. 8, 2017.
[12] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *International Conference on Information Security*. Heidelberg, Germany, 343–358.
[13] Zachary H Basnight. 2013. *Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers*. Wright-Patterson AFB, OH.
[14] Allen Bradley. 1996. *DF1 Protocol and Command Set, Reference Manual*. Milwaukee, WI.
[15] Allen Bradley. 2011. *MicroLogix 1100 Programmable Controller Instruction Set Reference Manual*. Milwaukee, WI.
[16] Allen Bradley. 2014. *MicroLogix 1100 Programmable Controller FRN 14*. Milwaukee, WI.
[17] Allen Bradley. 2016. *Logix5000 Data Access Programming Manual*. Milwaukee, WI.
[18] Sergey Bratus, Axel Hansen, and Anna Shubina. 2008. *LZfuzz: a fast compression-based fuzzer for poorly documented protocols*. Technical Report TR-2008 634. Darmouth College, Hanover, NH.
[19] Eric J Byres, Dan Hoffman, and Nate Kube. 2006. On Shaky Ground–A Study of Security Vulnerabilities in Control Protocols. (2006), 782–788.
[20] Ganesh Devarajan. 2007. Unraveling SCADA Protocols: Using Sulley Fuzzer. prestend at DEF CON 15. Las Vegas, NV.
[21] Stephen J Dunlap. 2013. *Timing-based Side Channel Analysis for Anomaly Detection in the Industrial Control System Environment*. Wright-Patterson AFB, OH.
[22] Matthew Franz. 2007. ICCP Exposed: Assessing the Attack Surface of the Utility Stack. In *Proceedings of SCADA Security Scientific Symposium*. Miami, FL.
[23] Roland Koch. [n. d.]. ([n. d.]). https://github.com/HSASec/ProFuzz Accessed March 25, 2016.
[24] Luis Mora. 2007. OPC Security White Paper. (January 2007). https://scadahacker.com/library/Documents/OPC_Security/OPC%20Security%20-%20OPC%20Exposed.pdf
[25] Inc. Open DeviceNet Vendor Association. 2008. *Communicating with RA Products Using EtherNet/IP Explicit Messaging*. Technical Report 1.2. ODVA, Inc., Ann Arbor, MI.
[26] Open DeviceNet Vendor Association, Inc. 2017. *The CIP Networks Library Volume 1: Common Industrial Protocol* (3.22 ed.). Open DeviceNet Vendor Association, Inc., Ann Arbor, MI.
[27] Open DeviceNet Vendor Association, Inc. 2017. *The CIP Networks Library Volume 2: EtherNet/IP Adaptation of CIP* (1.23 ed.). Open DeviceNet Vendor Association, Inc., Ann Arbor, MI.
[28] Aaron Portnoy, Pedram Amini, and Ryan Sears. [n. d.]. ([n. d.]). https://github.com/OpenRCE/sulley Accessed March 25, 2016.
[29] Xiong Qi, Peng Yong, Zhonghua Dai, Shengwei Yi, and Ting Wang. 2014. OPC-MFuzzer: A Novel Multi-Layers Vulnerability Detection Tool for OPC Protocol Based on Fuzzing Technology. *International Journal of Computer and Communication Engineering* 3, 4 (July 2014). http://search.proquest.com/docview/1618797232?pq-origsite=gscholar
[30] Rebecca Shapiro, Sergey Bratus, Edmond Rogers, and Sean Smith. 2011. Identifying Vulnerabilities in SCADA Systems via Fuzz-Testing. In *International Conference on Critical Infrastructure Protection*. Heidelberg, Germany, 57–72.
[31] Christopher Smith and Guillermo Francia III. 2012. Security fuzzing toolset. In *Proceedings of the 50th Annual Southeast Regional Conference*. Tuscaloosa, AL, 329–330.
[32] Keith A Stouffer, Joseph A Falco, and Karen A Scarfone. 2015. SP 800-82. Guide to Industrial Control Systems (ICS) Security: Supervisory Control and Data Acquisition (SCADA) systems, Distributed Control Systems (DCS), and Other Control System Configurations Such as Programmable Logic Controllers (PLC). (May 2015). http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r2.pdf
[33] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery* (1st ed. ed.). Addison-Wesley Professional, Boston, MA.
[34] Alexander Timorin. [n. d.]. ([n. d.]). https://github.com/atimorin/scada-tools Accessed March 25, 2016.
[35] A. Timorin. [n. d.]. Scada deep inside: protocols and security mechanisms. ([n. d.]). unpublished.
[36] Artemios G Voyiatzis, Konstantinos Katsigiannis, and Stavros Koubias. 2015. A Modbus/TCP fuzzer for testing internetworked industrial systems. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation*. Luxembourg City, Luxembourg, 1–6.
[37] Ting Wang, Qi Xiong, Haihui Gao, Yong Peng, Zhonghua Dai, and Shengwei Yi. 2013. Design and Implementation of Fuzzing Technology for OPC Protocol. In *2013 Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. Beijing, China, 424–428.
[38] Wireshark. n.d.. Code Review. (n.d.). https://code.wireshark.org/review/gitweb?p=wireshark.git; a=tree; f=epan/dissectors Accessed: 2016-08-18.