Automated Testing of SCADA Protocols

Nate Kube
Wurldtech Security Inc.
208-1040 Hamilton St., Vancouver, BC, Canada, V6B 2R9,
nkube@wurldtech.com

Daniel Hoffman
University of Victoria: Department of Computer Science
PO Box 3055 STN CSC, Victoria, BC, Canada, V8W 3P6,
dhoffman@cs.uvic.ca

Abstract: Testing is an important part of the Quality Assurance (QA) process, and the security portion of QA can detect potential vulnerabilities prior to exploitation. This paper begins by presenting background information on protocol testing and the available tools. We then discuss our experience in developing blackPeer, the testing framework for SCADA[1] protocols in Achilles, including lessons learned and key design decisions. The paper includes detail on the employment of blackPeer in vulnerability and conformance testing SCADA devices and the Achilles test methodology. We end with a discussion on common device errors and observations regarding the communication of these errors to the client.

Keywords: Vulnerability Testing, SCADA, Achilles, PLC

1 Introduction

The recent introduction of information technologies such as Windows, Ethernet and TCP/IP in industrial control devices has resulted in significantly less isolation from the outside world. Both anecdotal evidence [1] [2] and research [3] indicates that SCADA protocols, particularly those running over top of transport protocols such as TCP/IP, have vulnerabilities that could be exploited by network hackers or terrorists to cause considerable disruption to our critical infrastructures. Little is known about these vulnerabilities and there are limited security tools or methodologies available for vendors or users to detect these flaws prior to equipment deployment.

As highly integrated control systems are relatively new, there is shockingly little data, good or bad, on network security for these industrial devices. The current methodologies for security testing focus on business systems and their dependence on common operating system such as Windows and UNIX. Similarly, vulnerability reporting such as CERT or BugTraq primarily addresses IT products and rarely includes

---

[1] The term SCADA is used in this document to represent any industrial control system including Distributed Control Systems, Programmable Logic Controllers, Remote Terminal Units and Emergency Shut Down systems.

issues with industrial control products. In order to determine the security robustness of integrated control systems new testing methodologies are required.

This paper introduces and supports the need for new and efficient tools to test the network security robustness of industrial control devices. We briefly present background information on protocol testing and the tools which are available. We then introduce blackPeer, an innovative testing framework for communication protocols. We discuss its design and its evolution via field experience. We then present our test methodology, and end with a discussion on common device errors and our experiences disseminating this information to the client.

2 Background and Motivation

Many communication protocols are highly complex and their implementations are usually written to a specification that contains small but significant areas of ambiguity. Experience tells us that incorrect assumptions or carelessness of the implementer are common sources of protocol vulnerabilities. Protocol vulnerabilities can reveal themselves as segmentation faults, stack, heap or buffer overflows, etc., all of which can cause the protocol implementation to fail resulting in a potential exploit.

Tools to scan for known vulnerabilities in traditional IT systems have been available for at least a decade. The market for these vulnerability scanners has been significant and products such as Nessus, FoundScan and Internet Security Scanner (ISS) have been popular with IT administrators trying to locate unpatched computers on their networks. Unfortunately these tools offer little in the way of security testing for new products with new vulnerabilities; they only check for known vulnerabilities available in vulnerability lists such as CERT or BugTraq. As a result, most vendors have little knowledge of possible vulnerabilities in new systems until after the product is released to the public.

This is particularly true for the SCADA systems used in critical infrastructures such as the nuclear, oil and gas, water and electrical generation/distribution industries. The embedded devices used in these systems are not the usual Windows or UNIX-based platforms and the vulnerabilities are not available in the IT-focused CERT or BugTraq vulnerability lists. As a result, SCADA operators have little knowledge of possible vulnerabilities in their critical systems until a disaster (such as the August 2003 blackout in the Eastern U.S.) strikes.

In the academic world there have been several test tools that have had success in locating new vulnerabilities in network devices based on grammar and fuzzy techniques. Considerable work has been done by the PROTOS project group [4] and by Tal, Knight and Dean [5]. Each considers the syntax-based generation of *protocol data units* (PDU's). A PDU translates into a single test packet to be sent to the *device under test* (DUT). Their methods have proven effective in finding vulnerabilities [4][5]. However, they only allow for the construction of simple single-packet test cases. Such test cases are not sufficient for testing protocol functionality requiring interaction between the test case generator and the DUT. To perform such testing, the test case generator must be able to generate test cases consisting of semantically meaningful *sequences* of PDU's.

A sequence of PDU's in which each PDU is preceded by its communication direction relative to the test case generator is termed a *protocol test sequence* (PTS). PTS's can direct test case executors to not only transmit certain PDU's but also to compare received packets against indicated PDU's. Such ability greatly enhances the effectiveness of fuzzer-based methods.

Furthermore, as highly integrated control systems typically consist of many different devices and because these devices may contain implementations of many different protocols, a truly valuable protocol vulnerability testing tool must be easily applicable to a wide variety of protocols. As well, a valued tool must be employable by users with varying skill sets. For example, the tool should be employable by the vendor, by a field engineer or by a plant floor worker. To the best of our knowledge, no such tool exists.

3 Architecture and Design

The automated generation of test data for the purposes of software validation and verification has been a major aim of testing research[6]. Path-oriented methods, data flow methods, random testing, adaptive testing, and syntax-based testing are a few of the methods employed for this task.

Syntax-based testing tools process a description of the desired test data expressed in notation such as *Backus Naur Form* (BNF). Given the test data syntax, these tools generate test sets which satisfy the syntax. A major problem in syntax-based testing is respecting contextual dependencies. Solutions include the use of *dynamic syntax*, addition of rules to test data syntax during test data generation, and *attributed grammars*, grammars whose definition is overloaded with attributes.

In attribute grammars, the attributes represent contextual information associated with terminals and non-terminals. Attribute grammars can solve the test oracle problem as the contextual information can allow for the production of test data along with the output expected from the DUT. Hence, attribute grammars can also solve the problem of generating semantically meaningful sequences of PDU's, i.e., PTS's.

blackPeer is an attribute grammar based PTS generator and executor where PTS's are described by an attributed grammar. The initial file structure of blackPeer was similar of that employed by Sirer and Bershad [7] and is shown in Figure 1. In this structure, an attribute grammar is passed to the grammar_2_template file which parses the grammar, saving its productions to a template file. The template file is then merged with a tuple file which contains resolutions for each of the templates parameters. The resultant of the merge is a ground file which is passed to an executor.

An executor is a program that packages PTS as payloads into the appropriate protocol and sends them to the DUT. The particular executor you employ for a given protocol is always one layer beneath. For example, the executor for a MODBUS/TCP grammar would package PTS in TCP payloads.

Along with the ground file, the executor is passed a configuration file which provides the executor with protocol specifics necessary to communicate with the DUT. For example if the executor employed is an IP executor than the configuration file would contain items such as the DUT's IP address.

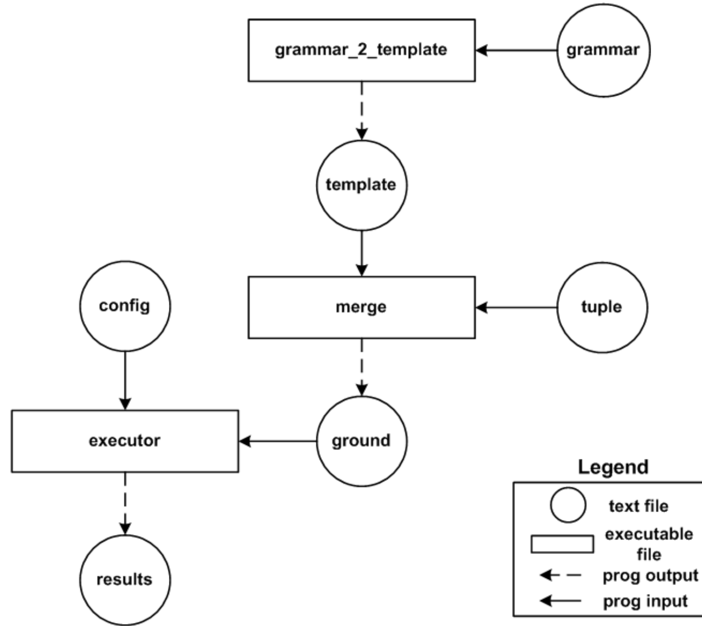

Figure 1 – Initial blackPeer Architecture

3.1 Minimal Example

Suppose that we wished to test the implementation of protocol X in the device under test by sending the one byte payloads 1, 11, 111. Suppose further that the protocol specification states the correct behavior is to send a one byte 0 back.

For this stateless test our grammar file may look like this:

```
%grammar{
        S::=SEND  EXPECT
        SEND::= TX  $TXDATA
        EXPECT::= RX  $RXDATA
}%
```

The template file would look like this:

```
TX $TXDATA RX $RXDATA
```

The tuple file would look like this:

```
1       0
11      0
111     0
```

The result of the merge would be:

```
TX 1 RX 0
TX 11 RX 0
TX 111 RX 0
```

The executor would read these test cases in from a file and one by one send out the bytes following the TX:0 and expect back those following the RX:0. For example, for test case 2 the executor would send out 11 in one byte and expect back 0 in one byte.

This design had several drawbacks. Many of the fields we wished parameterized were bound too early, forcing us to make changes to the grammar proper to adapt the tests to a new DUT. Furthermore, creating large tuple files was cumbersome and usually required the employment of an external program.

For these reasons and others we re-worked our design. We identified what we believed to be the important variability present between implementations of a given protocol, and we delayed their binding time as long as possible. The resulting design is shown in Figure 2.

The attribute grammar is passed as input to the *Code Generator*. The *Code Generator* parses the grammar and creates an executable program called the *TestCase_Generator*. When supplied with an initialization file called the *variable_ init* file, the *TestCase_Generator* writes out the PTS's encoded by the grammar to the *testCases* file. The *TestCase_Executor* then reads the *testCases* file along with basic information about the DUT and executes the PTS's one by one, reporting and recording the results.

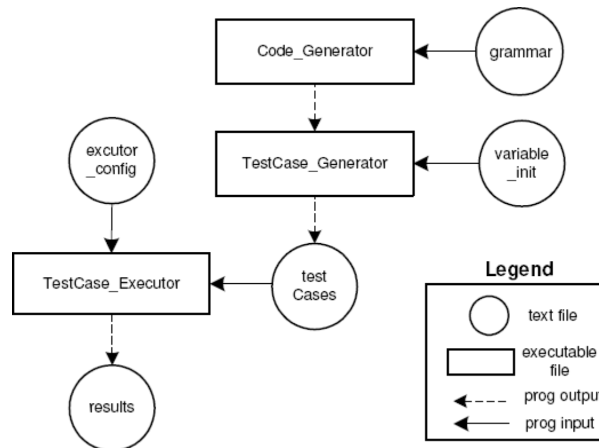

Figure 1 – Current blackPeer Architecture

3.2 Another Minimal Example

Once again, suppose we wished to test the implementation of protocol X in the device under test by sending the one byte payloads 1, 11, 111. Again, suppose the protocol specification states the correct behavior is to send a one byte 0 back.

For this stateless test our grammar file may look like this:

```
%grammar{

        S::=SEND  EXPECT
        SEND::= TX:0  DATA
        EXPECT::= RX:0  0:1
        DATA ::= 1:1 | 11:1 | 111:1
}%
```

The test cases produced by this grammar would be:

```
TX:0 1:1 RX:0 0:1
TX:0 11:1 RX:0 0:1
TX:0 111:1 RX:0 0:1
```

The executor would read these test cases in from a file and one by one send out the bytes following the TX:0 and expect back those following the RX:0.

The modular nature of this blackPeer design allows different levels of functionality to be made available to different categories of users. For example, a plant worker who knows little of communication protocols but much about PLC's may be given a *testCases* file and an appropriate *testCase_Executor*. An industrial engineer on the other hand may be given the *testCase_Generator* thus allowing him to create a variety of *testCases* files depending on the nature of the variable init file he creates.

This modular nature also allows for the easy transition between different protocols. That is, to test a different protocol all that is needed is a new grammar file and an appropriate *testCase_Executor*.

4 Achilles Test Methodology

No amount of testing can guarantee correct device behavior in the field. Only running all possible tests could do that, and there are generally far too many possible tests to exhaustively run them all. For instance, the cardinality of the test domain typically increases exponentially with the addition of each new variability, a property commonly referred to as combinatorial explosion. Worse, due to timing variations, a device may pass a test once and fail when the test is rerun later. The pioneering software engineer, E.W. Dijkstra, summarizes the situation well: “Program testing can show the presence of bugs but never their absence.”

Nonetheless, testing is the most common approach for finding bugs; a failed test definitively proves that a device has a bug. Well-designed tests can also exercise a device in near real-world conditions, demonstrating device capabilities and limitations, qualitatively and quantitatively. Such tests increase confidence in device performance,

even though absolute confidence is not achievable. Testing can provide valuable data to support comparisons between devices, including different versions of the same device and devices from different vendors. Finally, testing can provide legal and regulatory protection. As systematic testing of networked devices becomes common engineering practice, it will become increasingly risky to omit.

We now describe the characteristics of Achilles testing in terms of test plan, test methodology, test results, and test automation. An Achilles test plan describes the objectives of the testing, the features of the DUT, and the laboratory configuration. The test cases to be executed are described in detail.

Achilles testing covers all relevant layers of the protocol stack. The tests exercise both the protocols used directly by the DUT, e.g., TCP/IP and MODBUS, as well as utility protocols, e.g., ARP and ICMP. The test suite covers the many well-known network attacks, including those based on flooding the communications channel, consuming resources on the DUT, and inserting purposely illegal packet header fields. Since Achilles testing is based on formal methods, quantifiable claims regarding test coverage are possible. Finally, Achilles output checking uses, among others, device monitors; an OPC client continuously captures and evaluates device behavior during test case execution for example.

Achilles testing provides detailed results for each test case enabling the reproduction of the result at a later date. The results also include recommendations regarding device hardening and required switch and firewall configuration.

Finally, when test cases are numerous, manual execution and output checking is expensive, making effective testing unaffordable. With test automation, many tests can be run economically. Tests can also be repeated easily on another device or device version, or when the results on the original device are contested. With Achilles, test automation is pervasive: a test case is a parameterized Achilles plugin and a test suite is a customizable script invoking multiple test cases and monitors. Achilles test suites are easy to execute repeatedly and easy to reconfigure thereby enabling regression testing which is prevalent in any efficient quality assurance cycle.

5 Test Results

This section presents some of the more common controller vulnerabilities and non-conformance issues Achilles (using the blackPeer framework) has detected. Detected vulnerabilities are loosely categorized into one of three classes:

- **Critical:** the controller exhibited an internal fault and power cycled (akin to a hard reset);
- **Loss-of-view:** the communication between the vendor control software (HMI for example) and the device under test was eliminated; and
- **Non-critical:** the controller exhibited an error in expected functionality but did not experience a reset.

Common triggers of critical vulnerabilities are:

- Unicast, broadcast and multicast Ethernet storms
- Random Ethernet storms
- Derivatives of IP land attacks
- Unicast IP storms
- Minor attacks on proprietary application services

Common triggers of loss-of-view vulnerabilities are:

- < 10Mbs Ethernet or IP DoS's
- ARP floods
- unsolicited ARP replies
- Invalid IP fragments
- IP SYN floods
- Unicast, broadcast and multicast IP storms
- Minor attacks on proprietary application services
- MODBUS/TCP function code 8 sub-function code 2 requests

Common triggers of non-critical vulnerabilities are:

- Carefully crafted invalid packets at layers 2 through 7
- Minor attacks on proprietary application services

For examples of non conformance to a protocol's specification we will show a subset of MODBUS/TCP conformance tests conducted against two representative PLC's. The MODBUS/TCP grammar selected for this example examines the behavior of the DUT in response to valid and invalid function code requests.

The function codes tested by this grammar are:

- 20 Read File Record
- 21 Write File Record
- 22 Mask Write Holding Registers
- 23 Read / Write Multiple Registers
- 24 Read FIFO Queue
- Fuzzed Headers

The results of the tests on PLC X and PLC Y are shown in Table 1. We see that the two PLC's failed the majority of the MODBUS/TCP conformance tests.

	PLC X	PLC Y
Test cases	3433	3433
Passed	531	2171
Failed	2902	1262

Table 1 – Results

We now break down the failed tests by function code.

5.1 Function Code 20

Both PLC X and PLC Y returned incorrect error codes under the following circumstances:

1. when the byte count < 7 or greater than 245
2. when the reference type !=6
3. when the record number >10000 or the record number + register length >10000

5.2 Function Code 21

Both PLC X and PLC Y returned incorrect error codes under the following circumstances:

1. when the byte count < 7 or greater than 245
2. when the reference type !=6
3. when the record number >10000 or the record number + register length >10000

5.3 Function Code 23

PLC X incorrectly reports an error when:

1. the starting write address was valid and the starting write address + quantity of registers to write was within range and the quantity of registers was 120 or 121 (limit is 121)

PLC X returned incorrect error codes under the following circumstances:

1. when the starting write address was valid and the quantity of registers to write was 120 or 121 and the starting address + quantity of registers to write was out of range and the data was valid

PLC Y returned incorrect error codes under the following circumstances:

1. when the read starting address was valid but the quantity of registers was zero
2. when the read starting address was valid and the read quantity of registers was out of range and the read starting address+ read quantity of registers was valid or invalid
3. when read starting address was invalid and the read quantity of registers was zero or out of range
4. when the write starting address was valid and the write quantity of registers was zero
5. when the write starting address and the write quantity of registers was valid and the write quantity + write starting address was within range but the actual data length and the and the specified data length were not in agreement
6. when the write starting address was valid and the write quantity was invalid and the write quantity + write starting address was or was not within range and the actual data length and the specified data length were or were not in agreement

5.4 Fuzzed MODBUS Headers

When fuzzing the MODBUS header PLC X displayed the following incorrect behaviors:

1. invalid function codes over 70 caused PLC X to send a TCP Reset thereby terminating the communication
2. incorrectly specified MODBUS packet lengths caused PLC X to send a TCP Reset thereby terminating the communication

When fuzzing the MODBUS header PLC Y displayed the following incorrect behaviors:

1. incorrectly specified MODBUS packet lengths caused PLC Y to send a TCP Reset thereby terminating the communication

We see that the level of granularity provided by the blackPeer framework is quite high. Furthermore, the tests only took approximately 5 minutes to conduct and the analysis approximately 20.

6 Communicating the Results

Since testing cannot guarantee device correctness, testers must be careful about interpreting test results. Vendors and asset owners frequently ask us questions such as:

- My device passed your tests; what does that mean?
- My device was vulnerable to certain attacks; can it be protected?

In principle, there are two main ways to answer these questions:

1. We can point to other devices which passed the tests and had good field experience.
2. We can explain the tests and their results.

In practice, item (1) is extremely difficult to accomplish for device testing. It is hard to find multiple fielded devices which have all passed the same test suites. Further, data on field errors is rarely shared, for obvious reasons.

Item (2) is primarily a communication, as opposed to a technical, problem. Explaining automated tests is difficult. Thorough device testing often involves dozens of test types and thousands of tests. We see three main approaches:

1. A carefully designed taxonomy is essential. A detailed description of all the test inputs and logs is far too cumbersome for communication. Tests could be organized by protocol layer or by type, e.g., resource starvation tests versus protocol robustness tests. Many tests will be parameterized. For example a DoS attack on an Ethernet port might be parameterized by a range of frame rates and frame lengths. Test descriptions must indicate the parameter values.
2. Many devices will be vulnerable to some network attacks, especially DoS attacks at high rates. The test report should provide the basis for mitigation efforts. In particular, the test report should indicate clearly where off-the-shelf traffic shaping and packet filtering technologies will be effective.
3. Standardization of test types is critical. There is already some agreement but also a great deal of misunderstanding. In the device testing domain, there is nothing like RFC 2544 which has widely influenced Ethernet switch tests. A device testing standard may well emerge in the form of a widely used executable test suite.

7 Conclusions

The blackPeer test framework allows for the economical creation of powerful test suites capable of quickly and efficiently testing a protocol's implementation against its specification. The blackPeer test framework is superior to conventional testing methods in two key areas:

1. the framework not only automates the generation of test cases but also automates the interpretation of the DUT's behavior in response to these test cases. This is achieved by the novel approach of statefully generating a test case oracle in conjunction with each generated test case; and
2. the framework provides a formal language medium in which the tester can express a test suite. The grammar serves as a concise documentation of the test suite providing the tester the ability to make quantifiable claims regarding the nature of his testing.

The SCADA industry urgently needs to adopt better security robustness testing as standard practice. Industry bodies like the American National Standards Institute (ANSI) and the International Electromechanical Commission (IEC) need to mandate standardized security/conformance testing and certification for these critical devices. The number of errors routinely detected and the errors' significance shows that the security testing/certification of SCADA devices is critical to protect our national infrastructures from both accidental and deliberate attacks. As well as demonstrating the need for such testing, this paper also illustrates how it can be successfully conducted.

About the Authors – Dr. Daniel Hoffman received the B.A. degree in mathematics from the State University of New York, Binghamton, in 1974, and the M.S. and Ph.D. degrees in Computer Science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial programmer/analyst. He is currently a Professor of Computer Science at the University of Victoria (British Columbia, Canada). Dr. Hoffman's research emphasizes the industrial application of software documentation, inspection, and testing. He spent the 1992-93 year on sabbatical at Tandem Computers, Inc. working on software inspection and automated class testing, and the 1998-99 year on sabbatical at Bell Laboratories, Lucent Technologies doing research in software product lines and test automation. More recently, he has focused primarily on the automated testing of computer network hardware and software.

Nate Kube received the honors B.Sc. degree in Mathematics and Computer Science from the University of Victoria in 2002. With a national scholarship, he entered UVIC's PhD program in Computer Science. While in the PhD program he worked at BCIT's Critical Infrastructure Protection Lab with the acclaimed industrial security researcher Eric J. Byres. Among other industrial security projects, Mr. Kube played an integral role in the design and architecture of the Achilles project - an innovative vulnerability assessment tool for industrial controllers. Mr. Kube is currently defending his PhD work on vulnerability testing protocol implementations in SCADA/DCS devices. In 2006 Mr. Kube co-founded Wurldtech where he currently holds the position of CTO. He is responsible for the management of industrial security testing and product development operations including Wurldtech's flagship product Achilles. Mr. Kube and his team specialize in formal methods for software testing, operations research / statistical modeling, critical infrastructure protection and embedded systems design & testing.

References

[1] Vulnerability Note VU#190617: LiveData ICCP Server heap buffer, US Computer Emergency Response Team, May 16, 2006, http://www.kb.cert.org/vuls/id/190617

[2] D. P. DUGGAN, M. BERG, J. DILLINGER and J. STAMP; "Penetration Testing of Industrial Control Systems", Sandia National Laboratories, March 7, 2005.

[3] E.J. BYRES, J. CARTER, A. ELRAMLY and D. HOFFMAN; "Worlds in Collision: Ethernet on the Plant Floor", ISA Emerging Technologies Conference, Instrumentation Systems and Automation Society, Chicago, October (2002).

[4] R. KAKSONEN, M. LAASKO and A. TAKANEN, "Vulnerability analysis of software through syntax testing," University of Oulu, Finland, *Tech. Rep.* (2000).

[5] O. TAL, S. KNIGHT and T. DEAN, "Syntax-based vulnerability testing of frame-based network protocols," *Privacy, Security and Trust* (2004).

[6] D. INCE, "The automatic generation of test data," *The Computer Journal*, **30**, 1 (1987).

[7] E. G. SIRER and B. N. BERSHAD, "Using production grammars in software testing," *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, New York, NY, pp 1-13, ACM Press, (1999).