Building Hardened Implementations of SCADA/ICS Protocols Using Language-Theoretic Security

Prashant Anantharaman, Dartmouth College

pa@cs.dartmouth.edu
http://cs.dartmouth.edu/~pa
CREDC Industry Workshop
March 27-29, 2017
Academia + Power Industry + InfoSec Community

• Sven M. Hallberg, Sergey Bratus, Adam Crain, Meredith L. Patterson, Maxwell Koo, Sean W. Smith
Outline

• Introduction: Parsers, security, and the LangSec viewpoint

• The Problem: Analysing the DNP3 protocol and vulnerabilities

• Our Approach: Building a safer DNP3 parser from scratch
  “Make the parser code look like the grammar”
  “Make the validity expectations of data apparent from code”

• Case study: a DNP3 filtering proxy

• Conclusion and next steps
Data format is code's destiny

• A **parser** takes input data, and builds a structural representation of the input
• "Validating input" is judging what effect it will have on code
What goes wrong in parsing

- Object boundaries in message cross or overlap
- Object are embedded in other objects incorrectly
- Objects that should appear in a given position aren’t there
- Objects appear in a position that isn’t right
- Pre-conditions expected by the rest of the code are not met
- Code’s behavior on input is *not predictable*
  - Buffer overflow, memory corruption, exploitation

Result: security hole!
LangSec: Mission assurance for parsers

• Formal language theory: The fundamental underlying science for this problem!
• Seeks to prevent recurring programmer errors in protocols by identifying and eliminating problematic syntax & ambiguity
• LangSec identifies protocol/syntax features that make security an uphill battle:
  - specification is ambiguous: programmers disagree
  - validity check is too hard for a programmer to get right
  - several sources of truth
  - too much context needed to judge an object as valid or invalid
Outline

• Introduction: Parsers, security, and the LangSec viewpoint

• The Problem: Analysing the DNP3 protocol and vulnerabilities

• Our Approach: Building a safer DNP3 parser from scratch
  “Make the parser code look like the grammar”
  “Make the validity expectations of data apparent from code”

• Case study: a DNP3 filtering proxy

• Conclusion and next steps
DNP3 issues are not theoretical

- 2013 to 2014 – Over 30 CVEs related to input validation with DNP3 implementations. ("Robus Master Serial Killer", Sistrunk & Crain, 2014)
- Out of dozens of implementations only a small few were defect-free.
- Low-defect implementations chose a very conservative subset of DNP3 features.
## Security Holes Exist!

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-2523</td>
<td>The <code>dnpc_assert_object</code> function in <code>epan/dissectors/packet-dnp.c</code> in the DNP3 dissector in Wireshark 1.12.x before 1.12.10 and 2.0.x before 2.0.2 allows remote attackers to cause a denial of service (infinite loop) via a crafted packet.</td>
</tr>
<tr>
<td>CVE-2015-8214</td>
<td>Siemens SIMATIC CP 343-1 Advanced devices before 3.0.44, CP 343-1 Lean devices, CP 343-1 devices, TIM 3V-IE devices, TIM 3V-IE Advanced devices, TIM 3V-IE DNP3 devices, TIM 4R-IE devices, TIM 4R-IE DNP3 devices, CP 443-1 devices, and CP 443-1 Advanced devices might allow remote attackers to obtain administrative access via a session on TCP port 102.</td>
</tr>
<tr>
<td>CVE-2014-5429</td>
<td>DNP Master Driver 3.02 and earlier in Elipse SCADA 2.29 build 141 and earlier, E3 1.0 through 4.6, and Elipse Power 1.0 through 4.6 allows remote attackers to cause a denial of service (CPU consumption) via malformed packets.</td>
</tr>
<tr>
<td>CVE-2014-5426</td>
<td>MatrikonOPC OPC Server for DNP3 1.2.3 and earlier allows remote attackers to cause a denial of service (unhandled exception and DNP3 process crash) via a crafted message.</td>
</tr>
<tr>
<td>CVE-2014-5425</td>
<td>IOServer before Beta2112.exe allows remote attackers to cause a denial of service (out-of-bounds read and master entry consumption) via a null DNP3 header.</td>
</tr>
<tr>
<td>CVE-2014-5410</td>
<td>The DNP3 feature on Rockwell Automation Allen-Bradley MicroLogix 1400 1766-Lxxxx A FRN controllers 7 and earlier and 1400 1766-Lxxxx B FRN controllers before 15.001 allows remote attackers to cause a denial of service (process disruption) via malformed packets over (1) an Ethernet network or (2) a serial line.</td>
</tr>
</tbody>
</table>
Vulnerabilities

Unsolicited Response

Group 1 Variation 0

4 byte start/stop

Sizeless?!

- infinite loop
- missing data
- integer overflow?
- accepts broadcast

From: Adam Crain, Chris Sistrunk “Project Robus, Master Serial Killer”, S4x14
## DNP3 frame format

<table>
<thead>
<tr>
<th>DNP3 Frame</th>
<th>Data Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td></td>
</tr>
</tbody>
</table>

### Header

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>Length</td>
<td>Link Control</td>
<td>Destination Address</td>
<td>Source Address</td>
<td>CRC</td>
</tr>
</tbody>
</table>
From DNP3 link layer to application layer

<table>
<thead>
<tr>
<th>A = Application Header</th>
<th>T = Transport Header</th>
<th>L = Link Header</th>
</tr>
</thead>
</table>

Diagram showing the transmission sequence from DNP3 link layer to application layer, with fragments and headers indicated.
Application layer: Sequences of DNP3 objects

4.2.2.1 General fragment structure

Request and response fragments have similar, but slightly different, structures (Figure 4-4).

![Diagram of fragment structure]

Figure 4-4—Fragment structure

Each fragment begins with an application header that contains message control information. This is true for all fragments regardless of whether they appear in single or multiple fragment messages.
It is not all grammar!

<table>
<thead>
<tr>
<th>Group</th>
<th>Variation</th>
<th>Subset levels</th>
<th>Request (outstation must parse)</th>
<th>Response (master shall parse)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>X X X — —</td>
<td>—</td>
<td>2A (READ)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>X X X — —</td>
<td>—</td>
<td>2A (READ)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>X X X — —</td>
<td>00, 01, 08, 129 (RESPONSE)</td>
<td>00, 01</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>X X X — —</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>X X X — —</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>X X X — —</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

A.23.1.2.3 Notes

Read requests and responses shall use qualifier codes 0x07 if an outstation receives this request, it implicitly indicates current time.

This object can be included in a write request. Write request value of 1 for this object. When an outstation receives it, wants to set the current time in the outstation.
Outline

• Introduction: Parsers, security, and the LangSec viewpoint

• The Problem: Analysing the DNP3 protocol and vulnerabilities

• Our Approach: Building a safer DNP3 parser from scratch
  “Make the parser code look like the grammar”
  “Make the validity expectations of data apparent from code”

• Case study: a DNP3 filtering proxy
LangSec methodology

• We **read through** the protocol specification and extracted all syntax related information from text

• Gathered all text information into a **grammar specification**

• Identified **problematic** and **ambiguous** syntax likely to cause errors

• Wrote our parser very closely following the grammar
Solve language problems with a language approach

• Start with a grammar
  • If you don’t know what valid or expected syntax/content of a message is, how can you check it? Or interoperate?
  • If the protocol comes without a grammar, you need to derive one. It’s the only way! :(

• Write the parser to look like the grammar: succinct & incrementally testable

• Don’t start processing before you’re done parsing
The Recognizer design pattern

Language grammar

Input

Recognizer for the language

Only semantically valid input passes

Processing only well-typed objects and no raw input

Invalid, malformed and crafted packets
Parsing & protocol anti-patterns

• “Shotgun parsers”: input validity checks *intermixed* with processing code; no clear separation boundary
  • OpenSSL’s Heartbleed, GNU TLS Hello bug, ...

• Unnecessarily complex syntax (e.g., *context-sensitive* where *context-free* or *regular* would suffice)
  • Objects’ interpretation & legality depends on sibling object contents

• Parser *differentials* (parsers disagree about message contents)
  • X.509 CA vs client bugs, Android Master Key bugs, ...
Parser combinators: a natural choice

- **Hammer** parser construction kit: C/C++
  - Bindings for Java, Python, Ruby, .NET, Go
  - Three algorithmic parsing back-ends

- Freely available on GitHub:
  https://github.com/UpstandingHackers/hammer
Parser combinators at a glance (1)
Parser combinators at a glance (2)

```
05 64 14 F3  start  =  h_token("\x05\x64");
01 00 00 04  len  =  h_int_range(h_uint8(), 5, 255);
0A 3B C0 C3  ctrl  =  h_uint8();
01 3C 02 06  dst  =  h_uint16();
3C 03 06 3C  src  =  h_int_range(h_uint16(), 0, 65519);
04 06 3C 01  crc  =  h_uint16();
06 9A 12  hdr  =  h_attr_bool(h_sequence(h_ignore(start),
   len, ctrl, dst, src, crc, NULL),
   validate_crc);
frame  =  h_attr_bool(h_sequence(hdr,
   h_optional(transport_frame),
   h_end_p(), NULL), validate_len);
```
Parser Combinators: code looks like the grammar

• Have primitives

```c
HParse *seqno = h_bits(4, false);
HParse *bit = h_bits(1, false);
...
```

• Combined to form higher-level structures

```c
h_choice, h_many, h_many1, ...
```

• define own combinators
Fragment Header Flags - dealing with syntax overloading across 4 types of packets

/* --- uns,con,fin,fir --- */
conflags = h_sequence(bit,zro,one,one, NULL); // CONFIRM
reqflags = h_sequence(zro,zro,one,one, NULL); // always fin,fir!
unsflags = h_sequence(one,one,ign,ign, NULL); // unsolicited
rspflags = h_sequence(zro,bit,bit,bit, NULL);

Checking is declarative, not in code!
CROB Object

Declarative approach: parser code is generated from it.

crob = h_sequence(h_bits(4, false), // op type
               bit,       // queue flag
               bit,       // clear flag
               tcc,
               h_uint8(), // count
               h_uint32(), // on-time [ms]
               h_uint32(), // off-time [ms]
               status,    // 7 bits
               dnp3_p_reserved(1),
               NULL));
LangSec approach makes you ask the right questions

```
pcb = dnp3_p_g12v2_binoutcmd_pcb_oblock;
pcm = dnp3_p_g12v3_binoutcmd_pcm_oblock;
select_pcb = h_sequence(pcb, h_many1(pcm), NULL);
select_oblock = h_choice(select_pcb,
                        dnp3_p_g12v1_binoutcmd_crob_oblock,
                        dnp3_p_anaout_oblock,
                        NULL);
select = h_many(select_oblock);
```

// empty select requests valid?
// is it valid to have many pcb-pcm blocks in the same request?
// ... to mix pcbs and crobs?
// langsec approach warns you of pitfalls!

Failure to ask to ask any of these questions could result in parser bugs!

Parser bugs => 0days
Outline

• Introduction: Parsers, security, and the LangSec viewpoint

• The Problem: Analysing the DNP3 protocol and vulnerabilities

• Our Approach: Building a safer DNP3 parser from scratch
  “Make the parser code look like the grammar”
  “Make the validity expectations of data apparent from code”

• Case study: a DNP3 filtering proxy

• Conclusion and next steps
Practical application: Validating Proxy

Master  Bi-directional TCP Streams  Outstation

Dissector #1

Dissector #2

UC-8100 Series

Communication-centric RISC computing platform
Validation: tools & techniques

- Unit tests, Unit tests, Unit tests! (easy for parser combinators)
- Tests based on common DNP3 implementation mistakes
- Dynamic analysis with Valgrind
- Fuzzing: coverage-guided (AFL) and model-based (Aegis)
We survived AFL and Aegis!

- **American Fuzzy Lop (AFL):** Generic coverage-guided fuzzing (needs source)

- **Aegis:** Specialised DNP3 fuzzer by Adam Crain
Unit tests for smallest parser parts, known poison

// 4-byte max range - start = 0, stop = 0xFFFFFFFF
check_parse(dnp3_p_app_response,
    "\x00\x81\x00\x00\x1E\x02\x00\x00\x00\x00\xFF\xFF\xFF\xFF", 15,
    "PARAM_ERROR on [0] RESPONSE");

static HParsedToken *act_range(const HParseResult *p, void *user)
{
    // p->ast = (start, stop)
    uint32_t start = H_FIELD_UINT(0);
    uint32_t stop  = H_FIELD_UINT(1);

    assert(start <= stop);
    assert(stop - start < SIZE_MAX);
    return H_MAKE_UINT(stop - start + 1);
}
Write tests as you write production code

// mixing CROBs, analog output, and PCBs

check_parse(dnp3_p_app_request, 
    "\xC3\x03\x0\x02\x07\x01\x41\x03\xF4\x01\x00\x00\xD0\x07\x00\x00\x00"
    "\x0C\x03\x00\x05\x0F\x21\x04"
    "\x29\x01\x17\x01\x01\x12\x34\x56\x78\x00", 34,

"[3] (fir,fin) SELECT {g12v2 qc=07 (CLOSE PULSE_ON 3x on=500ms off=2000ms)}" 
  " {g12v3 qc=00 #5..15: 1 0 0 0 0 1 0 0 0 0 1}"
  " {g41v1 qc=17 #1:2018915346}"");
Outline

• Introduction: Parsers, security, and the LangSec viewpoint

• The Problem: Analysing the DNP3 protocol and vulnerabilities

• Our Approach: Building a safer DNP3 parser from scratch
  “Make the parser code look like the grammar”
  “Make the validity expectations of data apparent from code”

• Case study: a DNP3 filtering proxy

• Conclusion and next steps
Conclusion and next steps

• Langsec approach doesn’t guarantee success, but provides a disciplined roadmap for success

• Well-factored parsers will be more maintainable and extensible

• We are building parsers for ICCP and Modbus

• Our methodology makes the world a better place. There would be MORE 0 DAYS without this approach!
Thank you

LangSec talks & papers: http://langsec.org/

Code Availability:
- https://github.com/pesco/dnp3
- https://github.com/sergeybratus/proxy

Prashant - pa@cs.dartmouth.edu
Sergey Bratus - sergey@cs.dartmouth.edu
Sean Smith - sws@cs.dartmouth.edu
Funded by the U.S. Department of Energy and the U.S. Department of Homeland Security