SCL: A Language for Testing Security of Network Applications

Sylvain Marquis
Thomas Dean
Scott Knight

Royal Military College of Canada
Queen’s University
Software Developers are always under a great deal of time and budget constraints when developing software (Time to Market).

There are often implementation constraints
- memory footprint
- limited CPU performance
- demanding throughput requirements

Developer mindset is often on functionality and performance
- different viewpoint is necessary for security
Basic Premises

- Conformance tests for implementations of protocols tend to focus on correct responses to messages and obvious errors.
- Conformance tests typically do not test the security of the implementation.
- Implementations that meet conformance tests may have open vulnerabilities to more obscure cases that might not even be possible during the normal operation of the protocol.
Approach

- Protocols are complex systems.
  - It is a mistake to treat them only as data
  - Protocols have their own syntax and semantics
  - Semantics include constraints that must be implemented correctly

- Apply language analysis techniques to the protocols

- Generating test sets can be implemented using transformation tools and techniques
Syntax Testing

- Approach based on Bezier 90
  - When input has a syntax, then one appropriate test method is to break the syntax in various ways
  - Classified syntax testing into a set of errors that should be made

- Extended to network protocols by the PROTOS project at Oulu University (2001-Present)
PROTOS

- PROTOS uses a higher order attribute grammars to generate the packets
- The grammar specifies all elements down to test values for fields
- A walker generates test cases. Actions send packets, evaluate responses, calculate application specific info such as checksums.

<RRQ> ::= 0x00 0x01 <FileName><Mode>
<WRQ> ::= 0x00 0x02<FileName><Mode>
<Mode> ::= "netascii" 0x00
<FileName> ::= “sample.txt” 0x00
Scripting language is used
- Add, Replace actions used to augment or change the grammar to introduce errors
- Adds non-terminals from a library of grammar elements with predefined values for common protocol elements such as null terminated strings.
Our Test Framework

- Obtain a valid PDU from the network
Our Test Framework

- Obtain a valid PDU from the network
- Generate multiple mutant PDUs
Our Test Framework

- Obtain a valid PDU from the network
- Generate multiple mutant PDUs
- Inject mutant PDUs back into the network
Our Test Framework

- Obtain a valid PDU from the network
- Generate multiple mutant PDUs
- Inject mutant PDUs back into the network
- Between each mutant PDU, verify the health of the target with the valid PDU
**Mutation**

- Protocols have two levels
  - data structure (syntax/semantic) level
  - transfer encoding level

- Changes can be made at both levels

- Sniffing to generate the data provides a valid packet for the configuration
Good PDU

Decoder

Textual PDU

Test Plan

Script Generation

Script

Markup

Error Markup PDUs

Execution

Error PDUs (Text)

Encoder

Error PDUs (binary)
HDP_PDU ::= SEQUENCE {
  number_of_houses INTEGER
  houses               Houses
}

Houses ::= SEQUENCE OF House

House: ::= SEQUENCE {
  house_number      INTEGER
  family_name       VisibleString
}
HDP_PDU SEQUENCE {
    number_of_houses INT 2
    houses SEQUENCE {
        houses*1 SEQUENCE {
            house_number INT 200
            family_name VisibleString "Smith"
        }
    }
    houses*2 SEQUENCE {
        house_number INT 300
        family_name VisibleString "Stevens"
    }
}
HDP_PDU SEQUENCE {
    number_of_houses INT 2
    houses SEQUENCE {
        houses*1 SEQUENCE {
            house_number INT 200
            family_name VisibleString “Smith”
        }
        houses*2 SEQUENCE {
            house_number INT 300
            family_name VisibleString “Stevens”
        }
    }
}
HDP_PDU SEQUENCE {
  number_of_houses INT 2
  houses SEQUENCE {
    houses*1 SEQUENCE {
      house_number INT ErrASN remove 200
      family_name VisibleString “Smith”
    }
    houses*2 SEQUENCE {
      house_number INT 300
      family_name VisibleString “Stevens”
    }
  }
}

HDP_PDU SEQUENCE  ErrDER increase_length 400 {
    number_of_houses INT 2
    houses SEQUENCE {
        houses*1 SEQUENCE {
            house_number INT  ErrASN remove 200
            family_name    VisibleString "Smith"
        }
        houses*2 SEQUENCE {
            house_number INT 300
            family_name    VisibleString "Stevens"
        }
    }
}
Constraint Error

HDP_pdu SEQUENCE  ErrDER increase_length 400 {
    number_of_houses INT  ErrASN NewValue 3 2
    houses SEQUENCE {
        houses*1 SEQUENCE {
            house_number INT  ErrASN remove 200
            family_name VisibleString “Smith”
        }
        houses*2 SEQUENCE {
            house_number INT 300
            family_name VisibleString “Stevens”
        }
    }
}
HDP_PDU SEQUENCE  ErrDER increase_length 400 {

number_of_houses INT 3

houses SEQUENCE {

houses*1 SEQUENCE {

family_name VisibleString “Smith”

}

houses*2 SEQUENCE {

house_number INT 300

family_name VisibleString “Stevens”

}

}

}
Experience

- TFTP, SNMP
- X.509 Public Key Library
- OSPF

Framework is Protocol Independent, but human planning is needed
HDP_PDU ::= SEQUENCE {
   number_of_houses INTEGER
   houses             Houses
}

Houses ::= SEQUENCE OF House

House: := SEQUENCE {
   house_number    INTEGER
   family_name     VisibleString
}

This field gives the length of
Currently specified in textual prose in the ASN.1 Spec
HDP_PDU ::= SEQUENCE {
  number_of_houses INTEGER
  houses Houses
}

Houses ::= SEQUENCE OF House

House: ::= SEQUENCE {
  house_number INTEGER
  family_name VisibleString
}
Constraints

HDP_PDU ::= SEQUENCE {
    number_of_houses INTEGER
    houses          Houses
}

Houses ::= SEQUENCE OF House

House: ::= SEQUENCE {
    house_number    INTEGER
    family_name     VisibleString
}

The values are limited to 1-5000 inclusive

Also currently specified in textual prose in the ASN.1 Spec
HDP_PDU ::= SEQUENCE {
    number_of_houses  INTEGER
    houses            Houses
}

HDP_PDU ::= SEQUENCE {
    number_of_houses INTEGER
    houses Houses
}

<size>
    number_of_houses is 1 bytes
    houses is CONSTRAINED
</size>
SCL

HDP_PDU ::= SEQUENCE {
  number_of_houses INTEGER
  houses Houses
}
<size>
  number_of_houses is 1 bytes
  houses is CONSTRAINED
</size>
<transfer>
  CARDINALITY(houses) = number_of_houses
</transfer>
SCL

Houses ::= SEQUENCE of House
Houses ::= SEQUENCE of House
<size>
    House is SELFDEFINED
</size>
House: := SEQUENCE {
    house_number INTEGER
    family_name VisibleString
}
House: ::= SEQUENCE {
    house_number INTEGER
    family_name  VisibleString
}
<size>
    house_number is 2 bytes
    family_name is 10 bytes
</size>
House: := SEQUENCE {
    house_number    INTEGER
    family_name    VisibleString
}
<size>
    house_number is 2 bytes
    family_name is 10 bytes
</size>
<constraints>
    VALUE(house_number) = 1..5000
</constraints>
SCL

HDP_PDU ::= SEQUENCE {
    number_of_houses  INTEGER
    houses            Houses
}
<size>
    number_of_houses is 1 bytes
    houses is CONSTRAINED
</size>
<transfer>
    CARDINALITY(houses) = number_of_houses
</transfer>
HDP_PDU ::= SEQUENCE {
    number_of_houses   INTEGER
    houses             Houses
}
<size>
    number_of_houses is 1 bytes
    houses is CONSTRAINED
</size>
<transfer>
    CARDINALITY(houses) = number_of_houses
</transfer>
<constraints>
    ORDER(houses) = ASCENDING USING(Houses.house_number)
</constraints>
SCL Constraints

- Value
- Range
- Enumerated
- Cardinality
- Length
- Order (Sorted)
- Unique
Mutator Framework

Good PDU → Decoder → Textual PDU → Protocol Description → Extractor

Protocol Description → Protocol Model

Protocol Model → Test Planning

Test Planning → Test Database

Test Database → Encoder

Encoder → Mutant PDUs (binary)

Mutant PDUs (binary) → Execution

Execution → Mutant Markup PDUs

Mutant Markup PDUs → Markup

Markup → Textual PDU

Textual PDU → Decoder
Protocol Model

- Similar to low level model in Program Comprehension
- AST + Types + Constraints
- Tuple Attribute (TA)
Mutator Framework

Diagram:

1. Good PDU
2. Decoder
3. Textual PDU
4. Protocol Description
5. Extractor
6. Protocol Model
7. Test Planning
8. Test Database
9. Script
10. Mutant Markup PDUs
11. Execution
12. Mutant PDUs (Text)
13. Encoder
14. Mutant PDUs (binary)
Mutator Framework

Good PDU

Decoder

Textual PDU

Protocol Description

Extractor

Protocol Model

Test Planning

Test Database

Markup

Mutant Markup PDUs

Script

Execution

Mutant PDUs (Text)

Encoder

Mutant PDUs (binary)
**Validation**

- **OSPF and BGP Protocols**
  - Verified that we can decode instance of each type of packet in both protocol
  - Manual check of facts extracted to test script
Currently Request-Response (OSPF, BGP, TFTP, SNMP, x509 Certs)
Constraints between PDUs
Persistent Elements
Conversations
Formal State Models - Not Needed!!!
Current Research

- State Dependent Protocols
  - Capture sequence of PDUs
  - Mutate last PDU in sequence
  - Augment SCL with intra PDU constraints
  - Generate Script that includes copying elements of responses to future packets
  - Replay sequence.
Current Research

- Text Based Protocols
  - Currently Binary Protocols
  - SMTP, HTTP, SOAP
- More Traditional Grammars
- Agile Parsing to Introduce Errors
  (Aspect Grammars??)
- Multiple Layers (Browser → Web Server → CGI/PHP/J2EE → Database)
Other Issues

- Code Analysis
- Test–to–Code
  - Many current techniques
  - Test data generated from a semantic constraint. Does this help us?
- Code–to–Test
  - Picking Values for Changes
  - Lightweight Analysis
Questions?