Redesigning Secure Protocols to Compel Grammatical Compliance Checking

Keith Irwin
Winston-Salem State University

How I Met LangSec

- Meredith Patterson did a talk at CCC in 2011
  - *The Science of Insecurity*
  - Gives a terrific overview of what LangSec is and why it matters
- Lays out two main goals for security
  - Input should be recognized before being processed
  - Inputs should always be parsed the same way in different implementations

LangSec Goals

1. No more shotgun parsers
2. Use protocols and file formats where well-formed inputs can be recognized
   - No input languages where recognition requires Turing machines
3. Build uniform parsers
   - Test grammatical equivalence in different implementations

Pragmatic Difficulties

- 1 & 3 are **implementation** issues
- It's easier to change protocols than implementations
  - You can argue “this is a good idea, we should do it this way” to a standards committee and win
  - It's much harder to convince every programmer “you should do it this way”
  - We can't control who gets to build implementations

Pragmatic Difficulties

- It's easier to spot broken protocols than broken implementations
  - Shotgun parsers work fine for well-formed input
    - So does not having any input validation at all
  - Differing parsers which are mostly compatible won't usually be noticed
    - Plus how can you compare parser implementations
  - For normal users, these are invisible errors until attacked

Goal

- Change protocols to compel implementers to
  - Do explicit recognition of inputs
  - Ensure that recognition is consistent across implementations

Basic Approach

- Software which
  - Does not validate its input
  - Validates its input incorrectly
- Should be obviously broken
  - Testing should be likely to find problem
  - Ordinary users should sometimes see it

Basic Approach

- When messages are validated compute a function G as a side effect
  - G is a little game we make the programmer play
  - Should be easy to compute when validating input
  - Should be painful to compute otherwise
- Use result of G as an encryption key
  - One some portion of the message or on some future message

Programmer Options

- Validate input & compute G
  - Everything is good
- Don't validate input or validate it incorrectly
  - No encryption key = obvious errors
  - Brute-force encryption key = very slow program
  - Compute G without validating = hard and pointless program to write

Programmer Options

- Note that this includes
  - Use library which validates input and generates G as a side effect
  - We are happy to push programmers towards standard libraries

Assumptions

- Implementers will basically try to implement the specified protocol
  - Do reasonable testing with well-formed inputs
  - Check that it interoperates with other implementations
  - Not be able to just use old version without G
- Implementors will not discard results of validation

G Functions

- Are defined per test or validation
- Should not have a value when validation fails
  - To make ignoring validation result less likely
- Should produce a range of values
- Should be robust against small mistakes in implementation

Simple Example

- Message $m=m_0 m_1$
- Check: $m_1 \neq 0$
- $G_{\neq 0}(m_1) = C / m_1$
  - where C is a very large constant
- New message = $E_{G_{\neq 0}(m_1)}\{m_0\}m_1$

Checking Regular Language Membership

- We have a more elaborate version of $G_{\neq 0}$ which we call Z
- Use this to build G_{FSA}
- Assume that protocol species an FSA to use for recognizing a regular language
  - Check that path through FSA ends in accept state
  - Check that path through FSA only contains valid edges

Checking Accept State

- Label all states in the FSA (function b represents label)
  - even numbers for accept states
  - odd numbers for non-accept states
- As we recognize a string and transition through $s_0 ... s_n$ we compute
  - $a(0) = b(s_0)$
  - $a(i) = 2a(i-1)+b(s_i)$

Checking Accept State

- A(FSA,m) = a(n)*j
  - j is the largest integer such that 2^j divides a(n)
  - m is message
- We'll later use A inside a Z function so we want A(FSA,m) = 0 when m is not accepted by FSA

Checking Valid Transitions

- Label all states and symbols with a prime number
- Calculate v for each state
  - Multiply the primes for all symbols which are not valid outgoing states
  - Multiply also the prime for the state itself

Checking Valid Transitions

- V(FSA,m)= $\sum_{0}^{n-1} v(s_i) \bmod p(\square_{i+1})$
- $G_{FSA}(m) = Z(A(FSA,m)*V(FSA,m))$
- If A or V is zero, $G_{FSA}(m)$ undefined

Checking Context Free Language Compliance

- Take advantage of fact that each rule in BNF is like a FSA
- Treat series of symbols for each BNF rule like an FSA
- Calculate A and V functions for them
- Multiply everything together
- Take Z function of the result

Conclusion

- We can change protocols to compel implementers to do language checks
- We can thus catch parsing errors even when only well-formed messages are exchanged
- Note: There may be better ways to build G functions for these checks

Mixing in Whole FSA

- To ensure that program is using correct FSA
- Make the b labels used in A by repeatedly choosing a transition and one of its end points
  - If end point is unlabeled, label it using the ongoing hash
  - If end point is labeled, rehash it
- Repeat until all states labeled

Mixing in Whole FSA

- If some transitions or states are missing
  - Other states will get wrong labels
  - A(FSM,m) will come out wrong
  - Even for messages which do not need those states or transitions for parsing

Z Function

- Undefined if i=0
- Assumes large constant *b*
- If programmer accidentally tries to pass 0 to second or third case, loops forever

$$Z(i)=\begin{cases} H(b) & if\ i=b \\ H(Z(2\mathrm{i})) & if\ 0<i<b \\ H(Z(\lfloor\sqrt{ib}\rfloor)) & if\ i>b \end{cases}$$