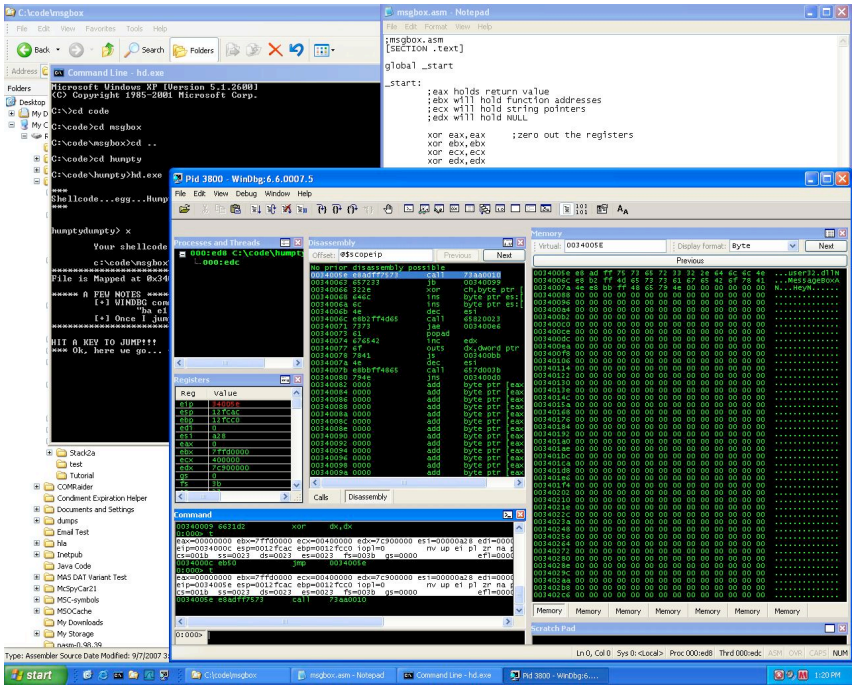
Fuzzing Frameworks, Fuzzing Languages?!

Stephen “sa7ori” Ridley, McAfee Senior Security Architect (former)
(DoD -> McAfee -> Independent RCE/Vuln Researcher)
stephen@blackroses.com

Colin Delaney, McAfee Software Security Engineer
(ActiveState -> MKS -> McAfee)
Colin_Delaney@mcafee.com

What is fuzzing”?

- Targeted application stress testing aimed at finding security flaws
- Supplying mangled data to a target application to stress parsers and data flow logic
- Modifying input anywhere that data is entering an application (files, registry, network, IPC, etc.)



Blind / Dumb fuzzing

- Non-”protocol aware”
- Corrupting random sections of input with random data
- Minimizes breadth and depth of testing
- Often quick and easy to use, but only capable of finding “shallow” bugs

```
1: \x00trValue="Hello World";
2: s\x00rValue="Hello World";
3: st\x00Value="Hello World";
4: str\x00alue="Hello World";
5: strV\x00lue="Hello World";
6: strVa\x00ue="Hello World";
7: strVal\x00e="Hello World";
8: strValu\x00="Hello World";
```

Smart (protocol aware) fuzzing

- Input protocol is replicated to support the fuzzing effort
- Fuzzer must be aware of the data types so that it can serve intelligent iterations
- Must understand how to return meaningful variations based on data types

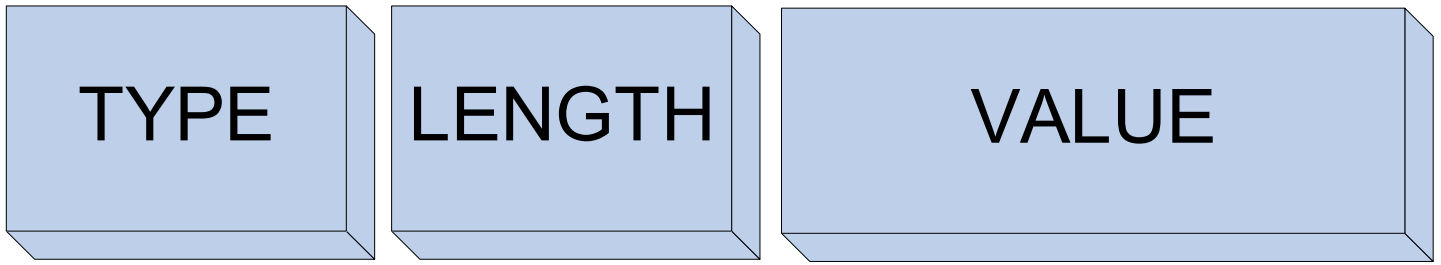
```
1: strValue="\x00ello World";
2: strValue="\xFFello World";
3: strValue="H\x00llo World";
4: strValue="H\xFFllo World";
5: strValue="He\x00lo World";
6: strValue="He\xFFlo World";
7: strValue="Hel\x00o World";
8: strValue="Hel\xFFo World";
```

Problems with "smart fuzzing"

- Manually replicating protocols is expensive
- Smart fuzzing highly targeted and therefore the code not easily reusable
- Improvements or innovations made to one fuzzing effort are not automatically available to other fuzzing projects without some sort of framework or object model

Dumb & smart fuzzing against a hypothetical protocol

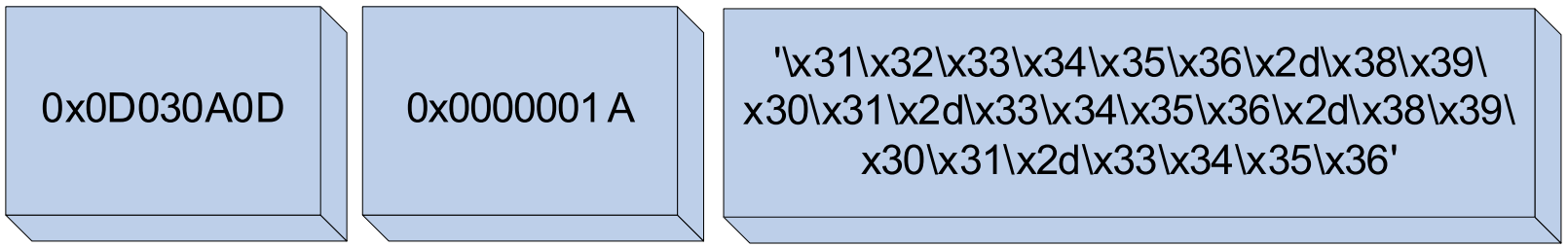
- Example: Type-Length-Value (TLV) protocol
  - Type: type field
  - Length: length of next section
  - Data: variable length data




```
0D030A0D  0000001A  123456-8901-3456-8901-3456
```

Example: dumb / blind fuzzing

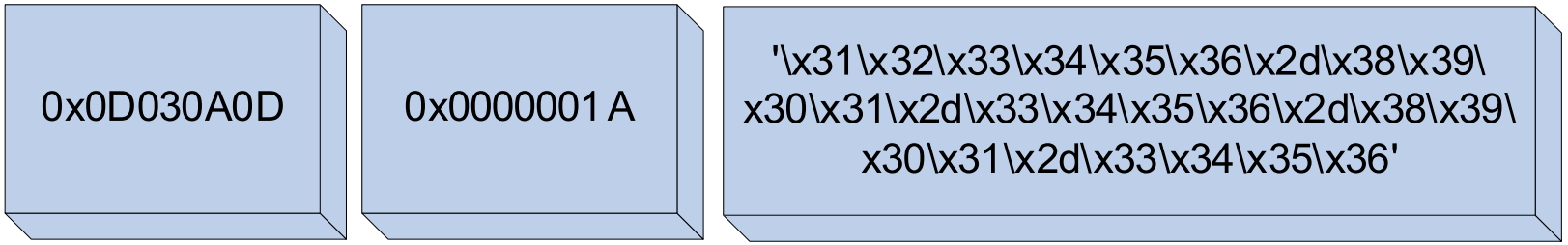
- A blind fuzzer will move through this block and randomly change bytes, characters, etc.
- Most random iterations will not conform to the protocol and therefore be discarded early
- Random mutation is unlikely to stress any of the fields in a meaningful way




```
~!030A0D   0000001A   123456-8901-3456-8901-3456
0D@#0A0D   0000001A   123456-8901-3456-8901-3456
0D03$%0D   0000001A   123456-8901-3456-8901-3456
```

Example: smart fuzzing

- Intelligent fuzzer is
  - Aware of the protocol
  - Returns meaningful iterations for all fields
  - Preserves fields we do not want to fuzz
  - Able to dynamically calculate lengths
- Each iteration is valid in regards to the protocol




```
0D030A0D   0000001A   FFFFFF-FFFF-FFFF-FFFF-FFFF
0D030A0D   00000001   0
0D030A0D   FFFFFFFF   AAAAAAAA[...]AAAAAAAAAAAAAA
```

Investigating
Current Fuzzing Solutions

- There are many fuzzers currently available, but we are going to limit our investigation to several of the main ones:
  - SPIKE
  - Sully
  - Peach

Investigating Current Fuzzers: SPIKE

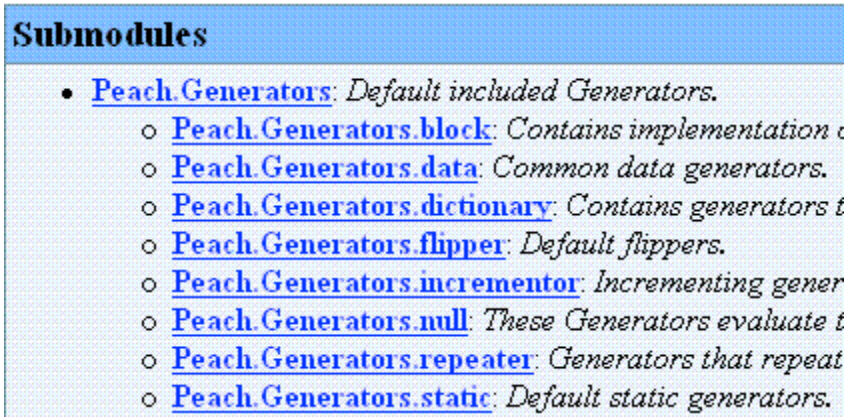
```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

- Pros
  - Widely used
  - Powerful
- Advantages of Ruxxer
  - Works on Windows and Linux
  - Does not require “C” coding knowledge
  - Object-model allows for improvements to be easily leveraged across fuzzing projects

Investigating Current Fuzzers: Sully

- Pros
  - API / Framework based
  - Debugger, target monitors
  - Code coverage metrics
- Advantages of Ruxxer
  - Stand-alone EXE that does not require a python installation
  - Abstracted, simple scripting language for less savvy users makes it more accessible
  - Experts can ignore the scripting language and work directly in “API mode” for added power

Investigating Current Fuzzers: Peach



- Pros
  - API / Framework
  - Extensible
- Advantages of Ruxxer
  - APIs are inherently not user-friendly, by adding a graphical Integrated Development Environment (IDE) on top of a framework, the tool becomes more easily consumable to a range of non-security aware engineers

A change in the fuzzing landscape (the evolution of fuzzing)

1. Blind Fuzzers
2. Mildly Protocol Aware Fuzzers
3. Fuzzing Frameworks for Protocol Awareness
4. Fuzzing Frameworks with basic Data Mutation
5. Frameworks with Data Mutation and Visualization
6. A complete Fuzzing Language with Data Visualization and Advanced Data Mutation capabilities

Rational for a New Approach

- Fuzzing have evolved:
  - Fuzzing now belongs in the Quality Assurance (QA) field as a pre-release testing activity
  - Engineering departments continue to incorporate aspects of the Security Development Lifecycle (SDL) into their Software Development Lifecycles (SDLC)
  - Security experts looking to stay on the edge need innovative tools that address their fuzzing “pains”

… A New Approach continued

- Engineers with limited security knowledge need to begin fuzzing their software suites on a tight budget

  … but …

- Security experts are not willing to compromise features for ease-of-use

… A New Approach: Compromise!

- APIs / Frameworks are extremely powerful but yield few immediate results because of the learning curve
- The compromise is to make the framework available via a simple scripting language that:
  - Provides experts with power
  - Innovates a new idea: a **Fuzzing Language!**

Introducing

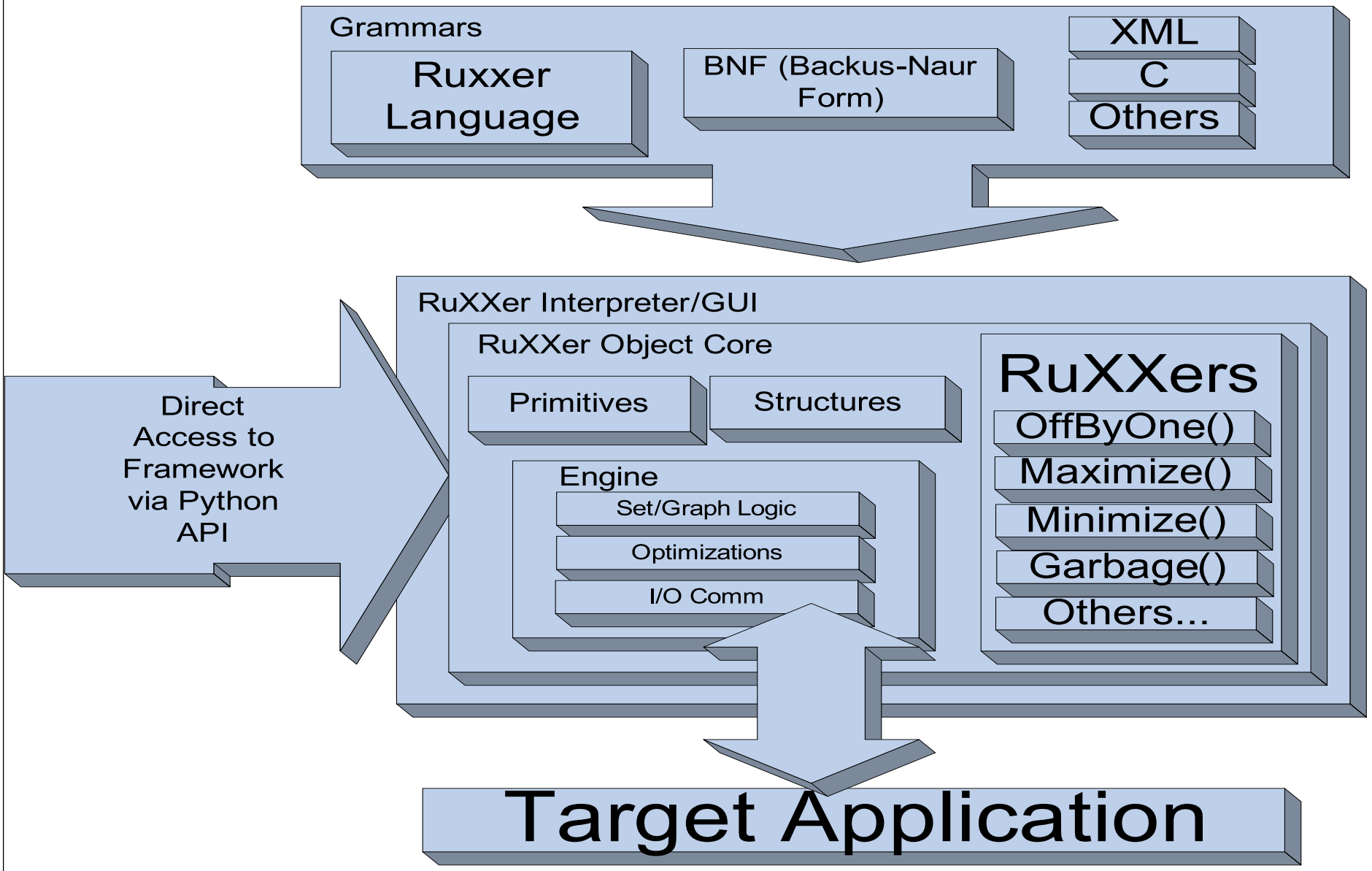
The RuXXer Architecture

- In short: RuXXer is a powerful Fuzzing Framework with its own language
- Composed of Two Major Pieces:
  - Language Interpreter
  - Fuzzing Framework Core
    - Object Model
      - Primitives, Structures, RuXXers
    - Data Mutator Engine

Demonstration
(whetting the palette)

- That's all well and good, but what does it look and feel like?
- If a Fuzzing Framework is an API with libraries then what comes with a "Fuzzing Language"?
- RuXXer application windows:
  - Coding Window
  - Data Visualization Window
  - Output Window

The RuXXer Architecture



Ruxxer Object Core

- An intuitive C-Like language
- (due to its object model) can also be used as a API/Framework directly from Python
- The Core centers around four concepts:
  - Primitives
  - Structures
  - RuXXers
  - Comms

Object Core: Primitives

- Most basic form of data.
  - like “legos” in Sulley
- Some basic RuXXer Primitives:
  - Byte
  - Short
  - Long
  - String
  - Length Calculators
  - Abstract Primitives (Email Address, CRC32, Hostname)

Object Core: Structures

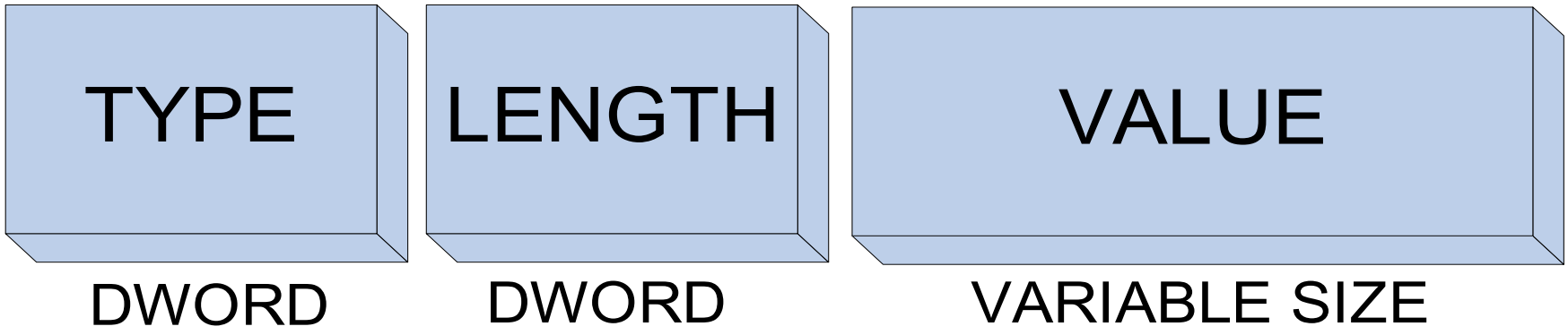
- Containers for Primitives
  - like “blocks” in Sulley
- Basic building blocks for abstract data types
- Logically similar to C “structs”
  - only less opaque
  - are actual instances

Object Core: Comms

- Delivers data to the target
  - TCPClient
  - TCPServer
  - UDPClient
  - UDPServer
  - FileOutput
- Easily extensible, more being added…

RuXXer Example: TLV Protocol

- Target: Hypothetical Protocol based on simple TLV (Type-Length-Value Protocol)
  - RIFF, PNG, etc.

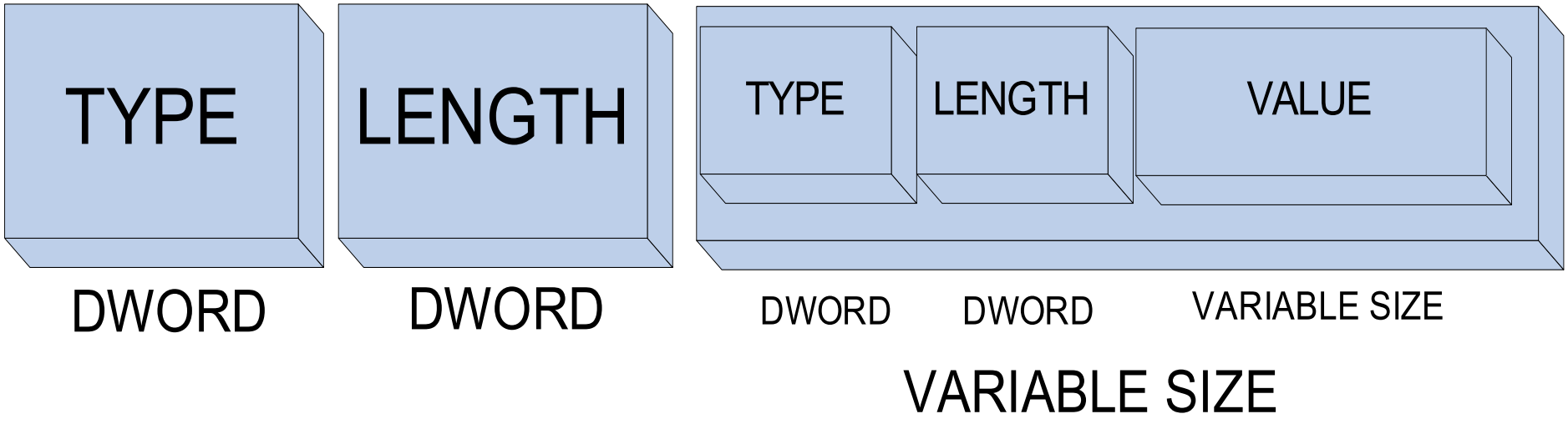


Modeling a TLV Protocol in Ruxxer

```
#declarations
long typeop;
long_lc pkt_len; #byte length calculator
string dgram;
structure tlv_packet;
#assignments
typeop = 0x0D030A0D;
pkt_len = tlv_packet;
dgram = “This is userdata\r\m”
push(tlv_packet, typeop, pkt_len, dgram);
```

Nested Protocols

- RuXXer's "Structure" types are designed to represent these kinds of complex nested data structures

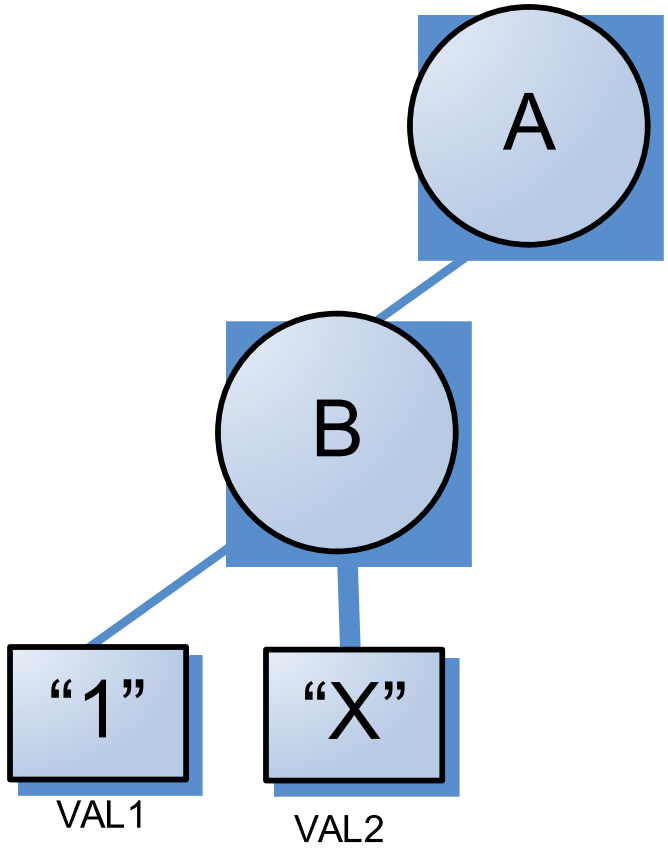


```
#declarations
long typeop;
long_lc pkt_len; #byte length calculator
structure dgram; # push(dgram, …, …, …
structure tlv_packet;

#assignments
typeop = 0x0D030A0D;
pkt_len = tlv_packet;
dgram = “This is userdata\r\m”
push(tlv_packet, typeop, pkt_len, dgram);
```

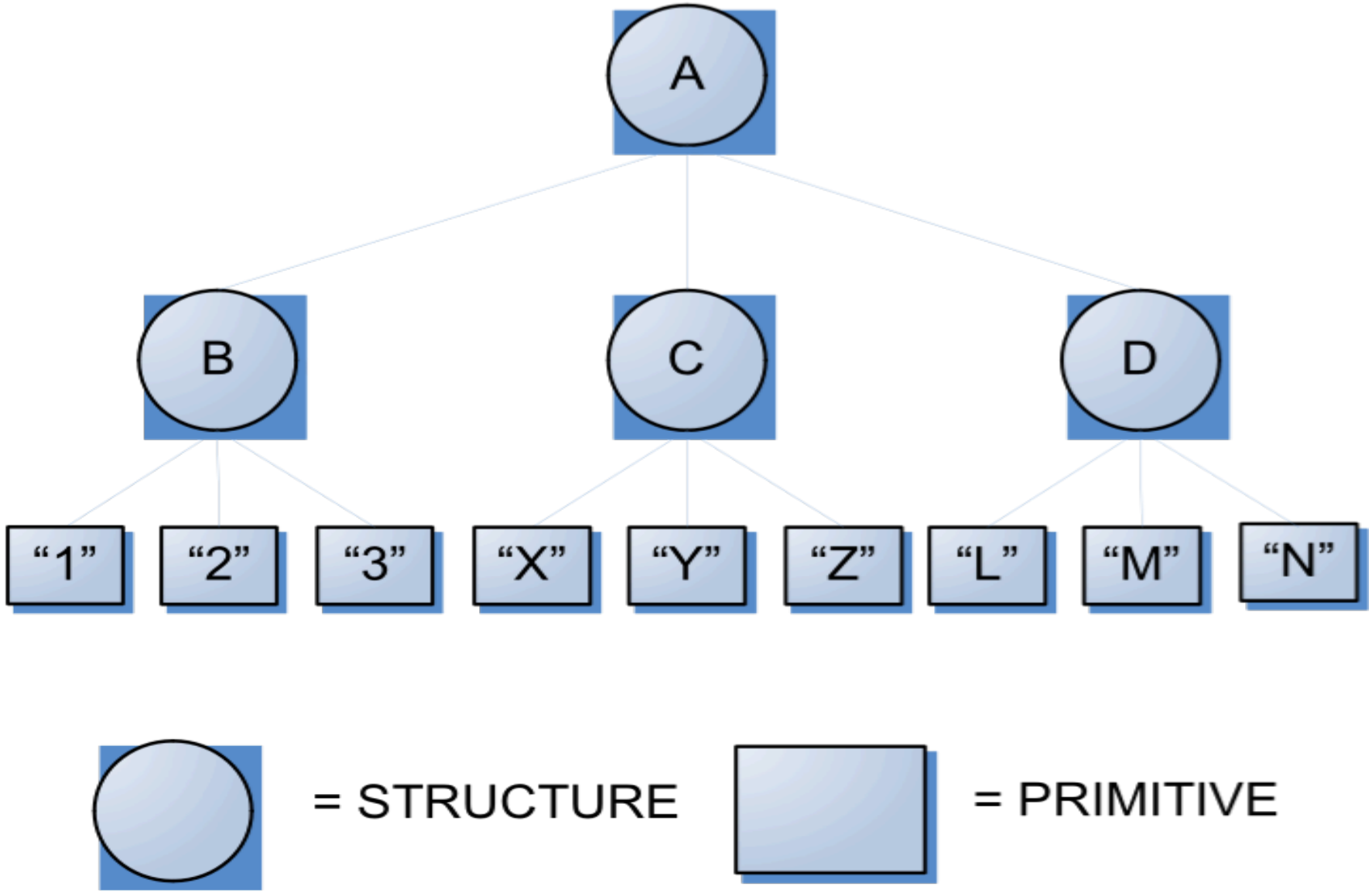
Graphical Representation

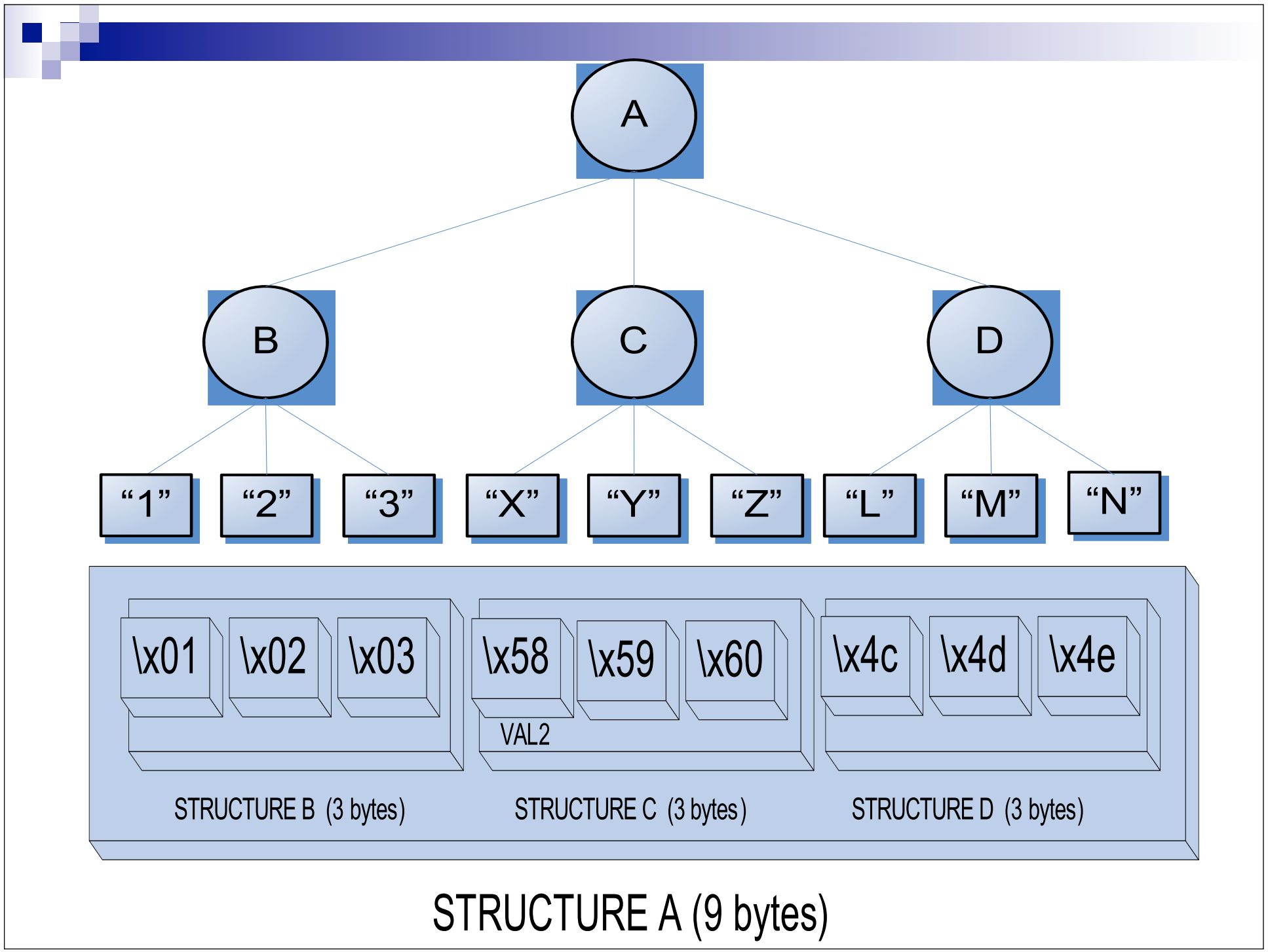
```
structure A;
structure B;
int val1;
val1 = 1;
string val2;
val2 = “X”;
#now we push
push(B, val1, val2);
push(A, B);
```

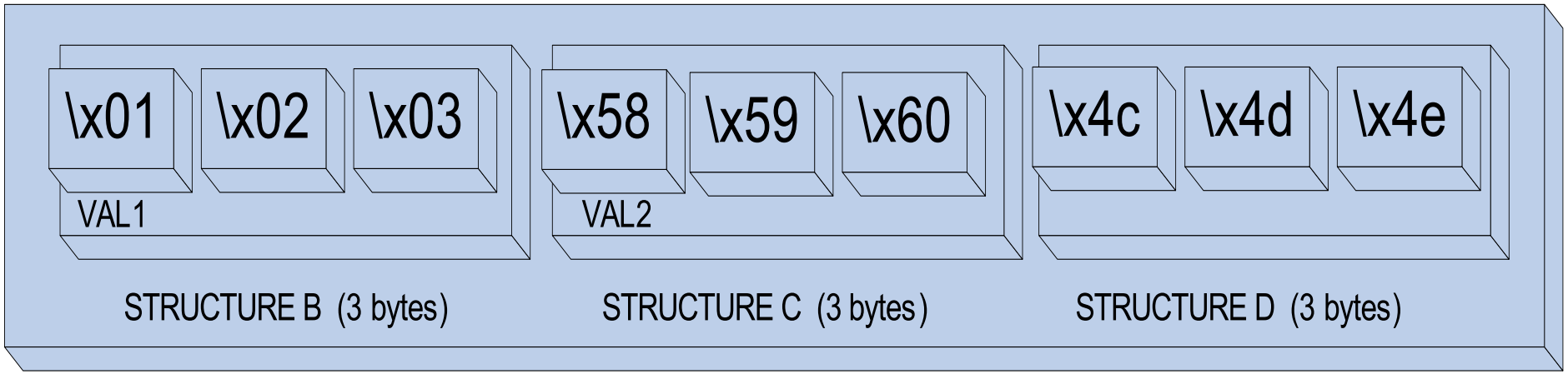


RuXXers: Intelligent Data Mutation

- RuXXers
  - change a Primitive's value in a specified way
  - form the basis for the concept of "reusable test cases"
  - are applied to specific Primitive types, eliminating pointless test cases
    - E.G. String tests run on numeric fields
  - Easily extensible, more being added all the time


A
B
C
D
“1”
“2”
“3”
“X”
“Y”
“Z”
“L”
“M”
“N”
= STRUCTURE
= PRIMITIVE


A
B
C
D
“1”
“2”
“3”
“X”
“Y”
“Z”
“L”
“M”
“N”
\x01
\x02
\x03
\x58
\x59
\x60
\x4c
\x4d
\x4e
VAL2
STRUCTURE B (3 bytes)
STRUCTURE C (3 bytes)
STRUCTURE D (3 bytes)
STRUCTURE A (9 bytes)



STRUCTURE A (9 bytes)

print(A);

>> “\x01\x02\x03\x58\x59\x60\x4c\x4d\x4e”

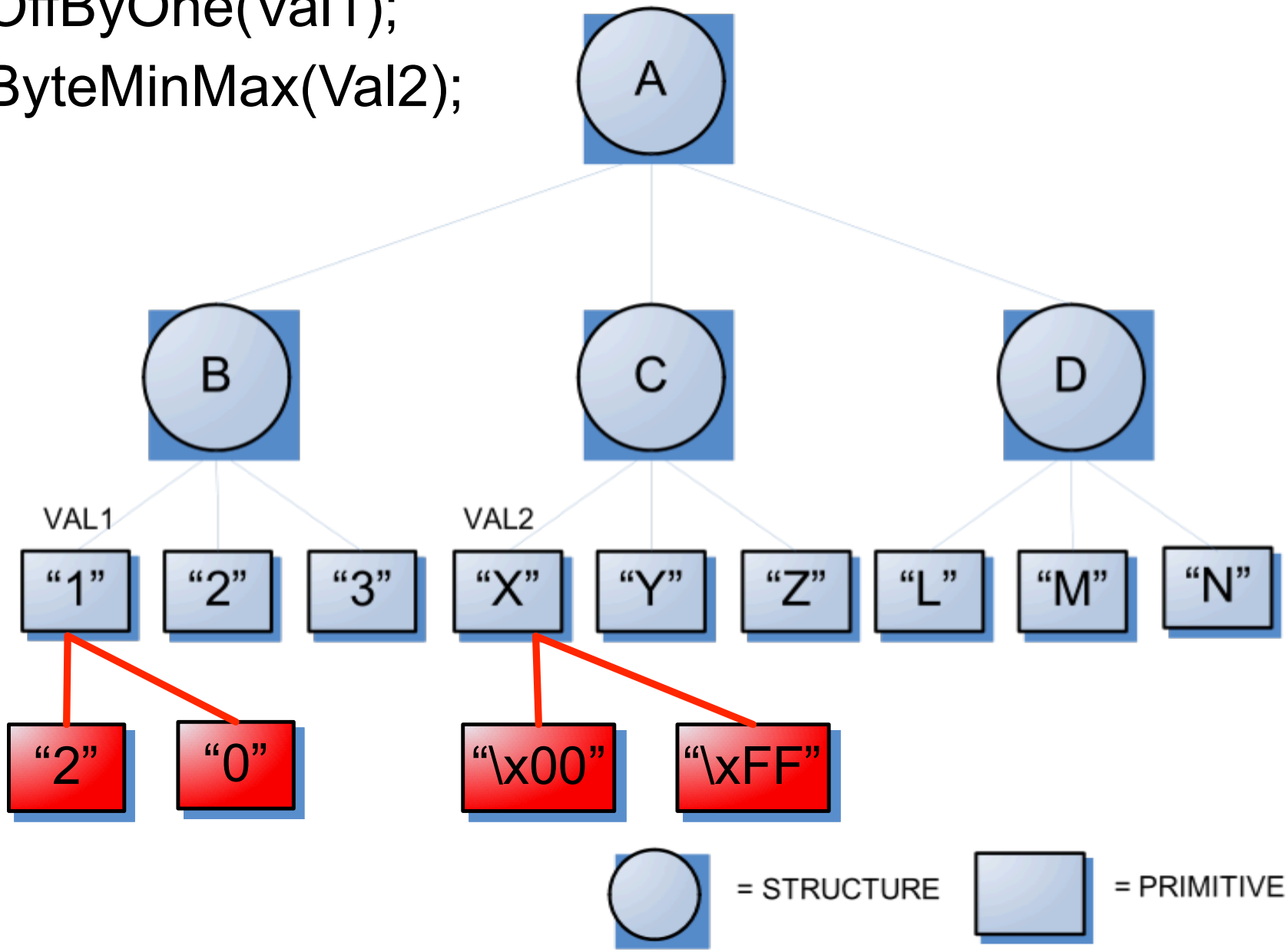
print(B);

>>“\x58\x59\x60”

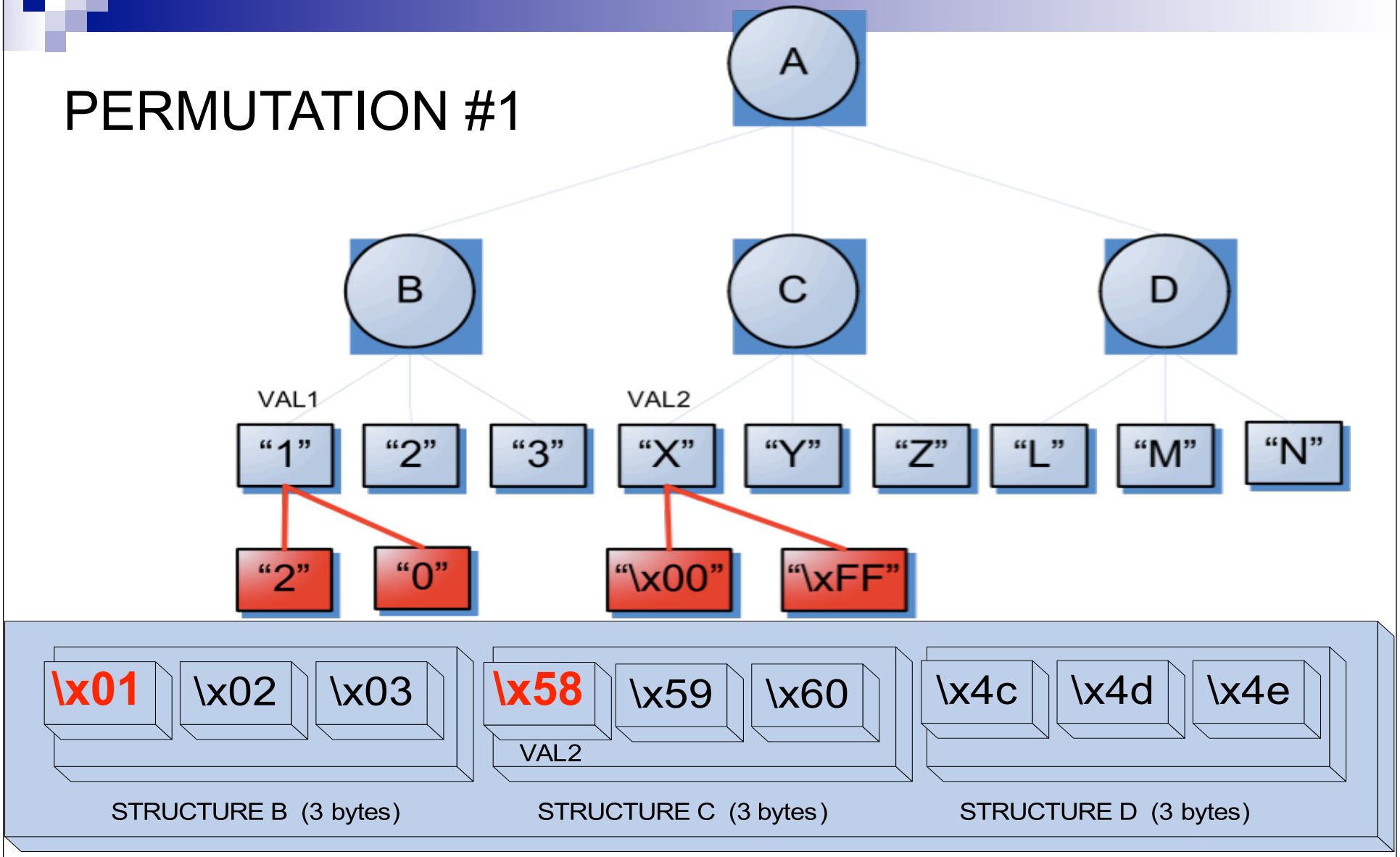
…..

OffByOne(Val1);

ByteMinMax(Val2);



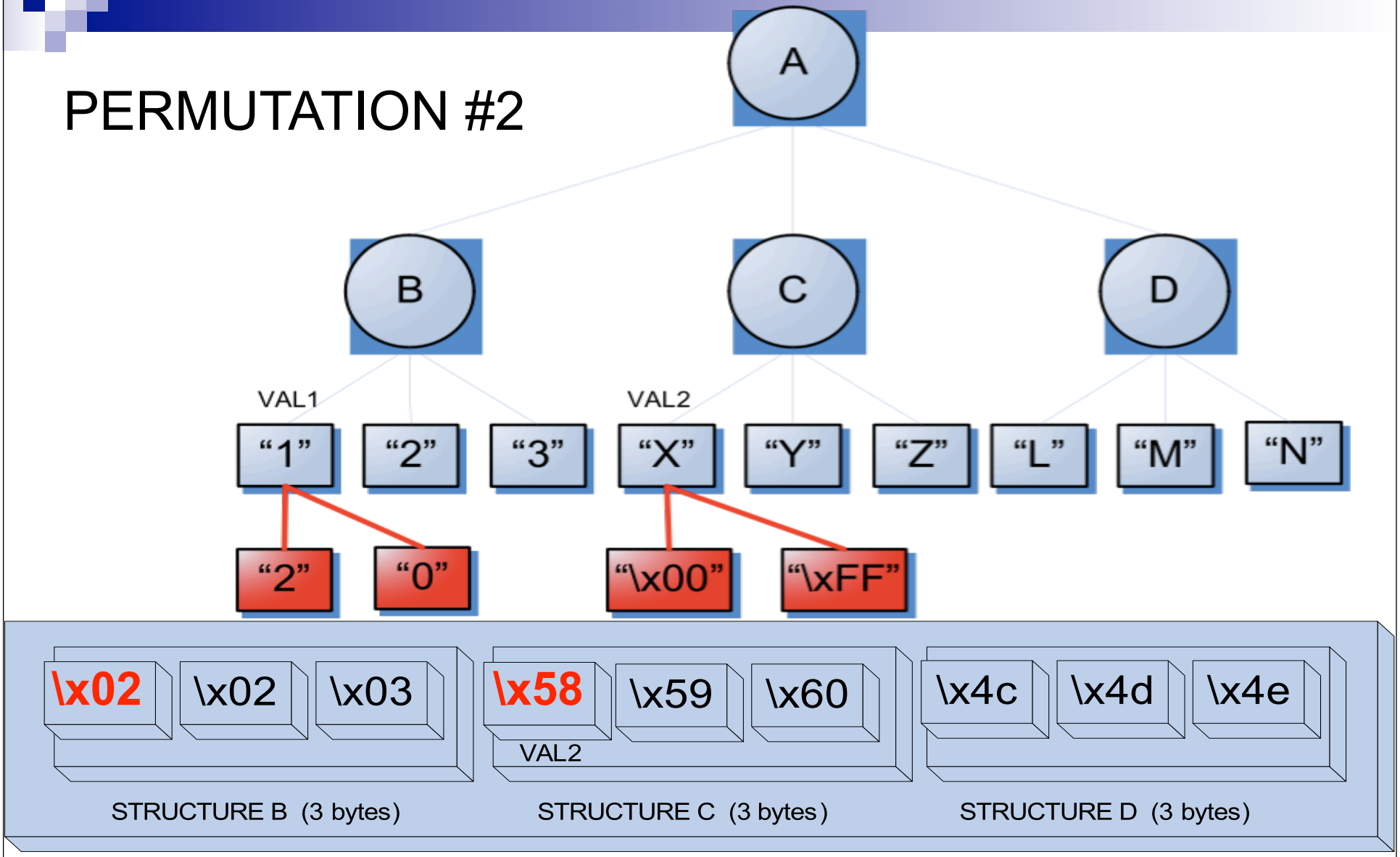
PERMUTATION #1



STRUCTURE A (9 bytes)

“\x01\x02\x03\x58\x59\x60\x4c\x4d\x4e”

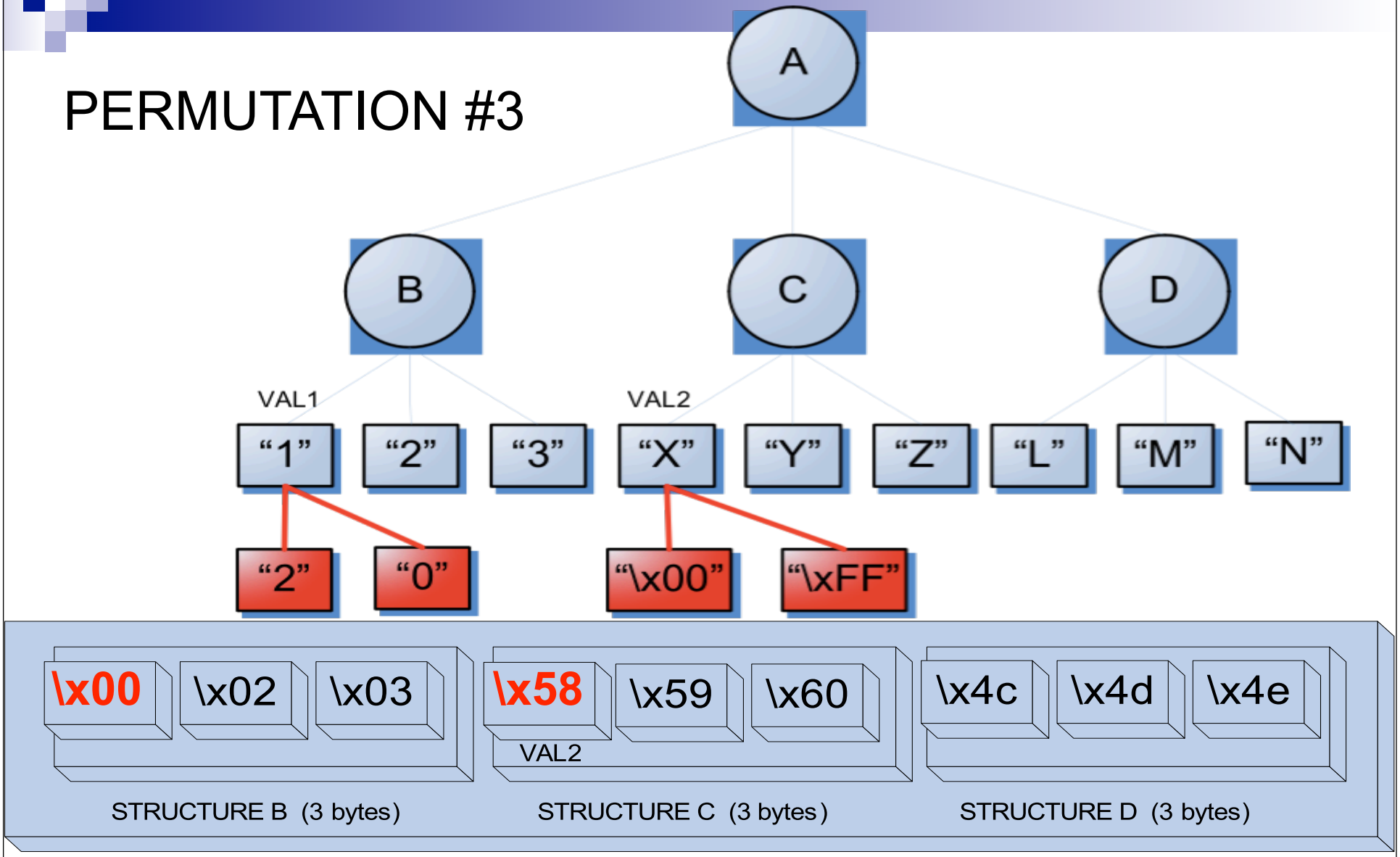
PERMUTATION #2



STRUCTURE A (9 bytes)

“\x02\x02\x03\x58\x59\x60\x4c\x4d\x4e”

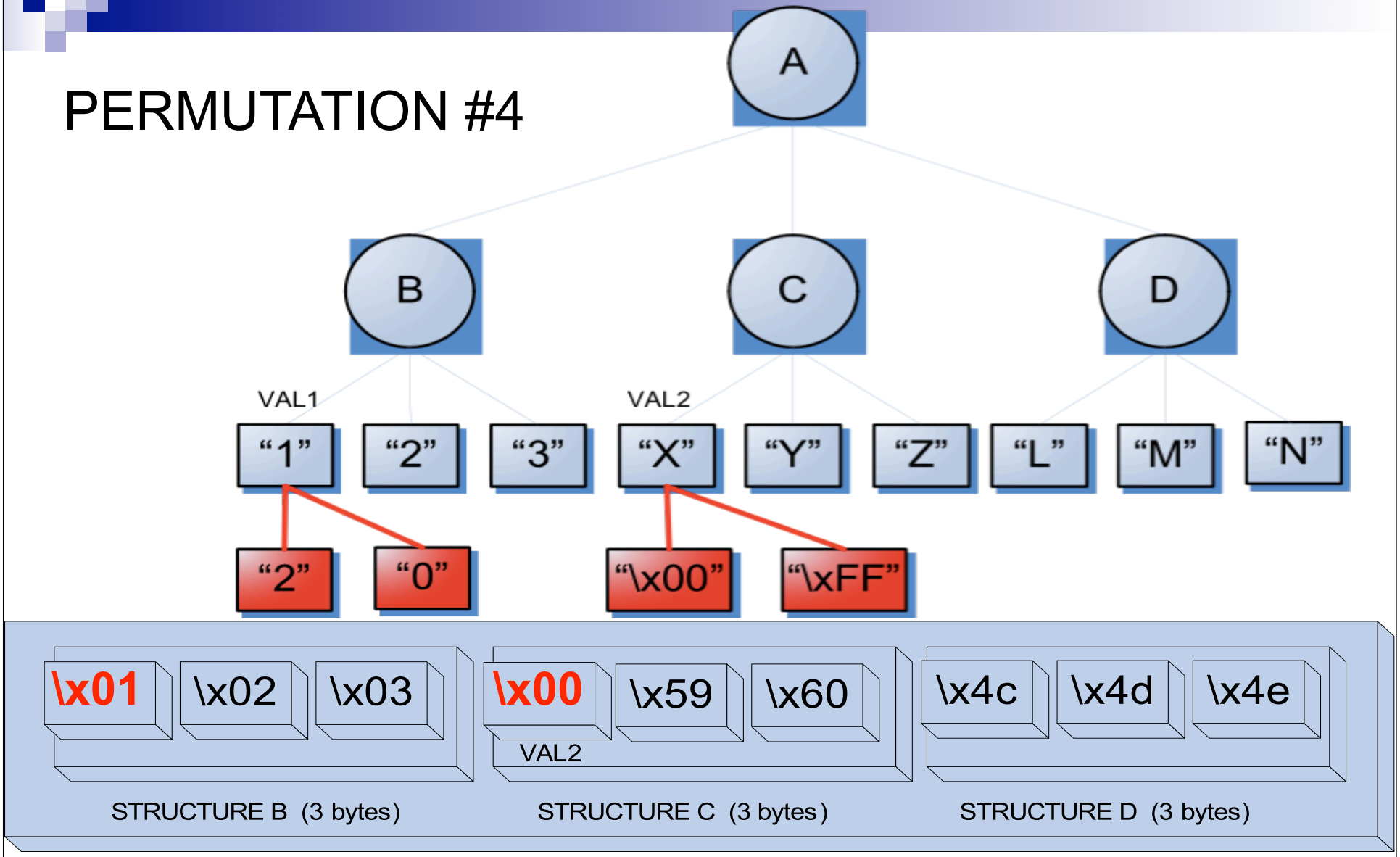
PERMUTATION #3



STRUCTURE A (9 bytes)

“\x00\x02\x03\x58\x59\x60\x4c\x4d\x4e”

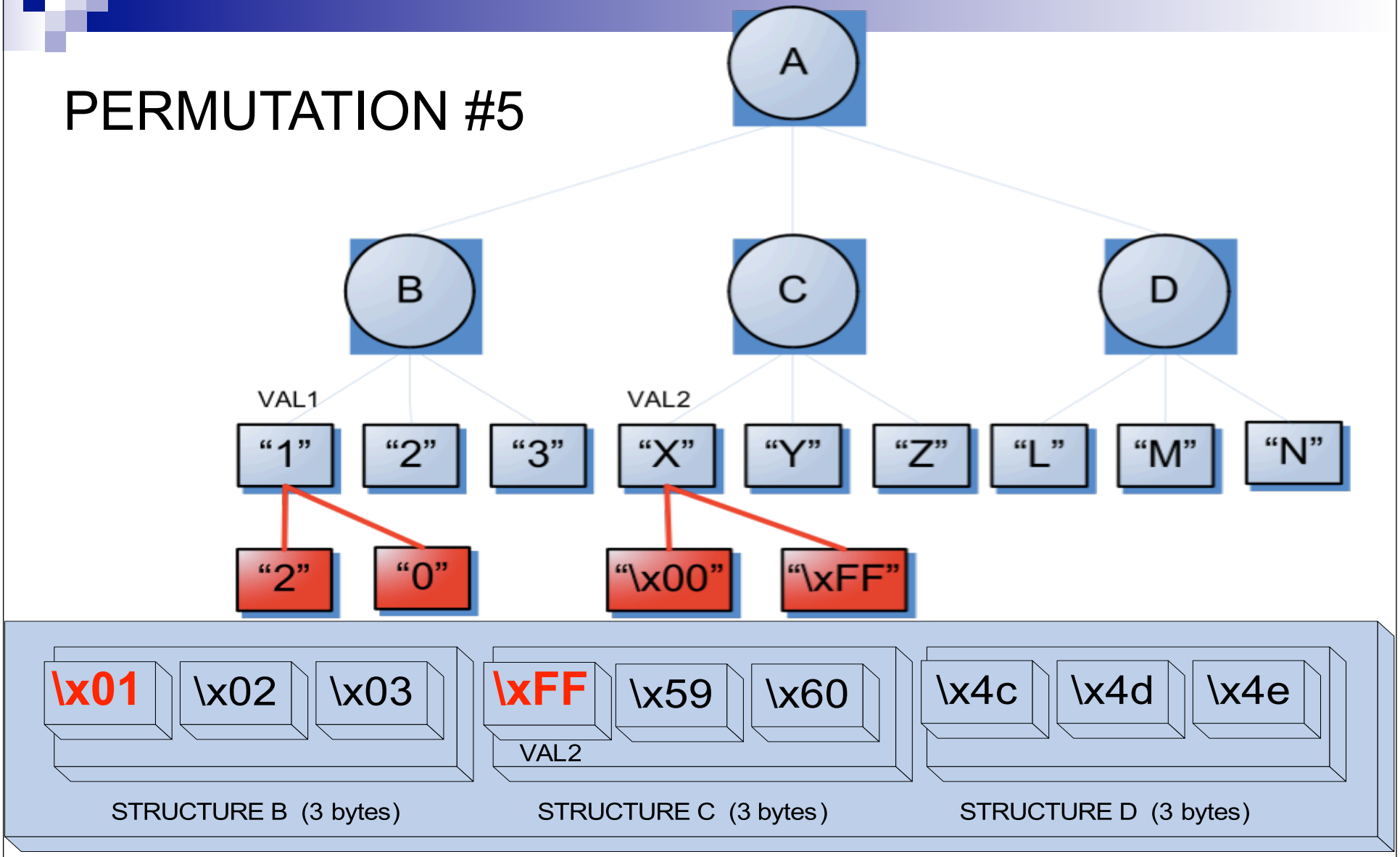
PERMUTATION #4



STRUCTURE A (9 bytes)

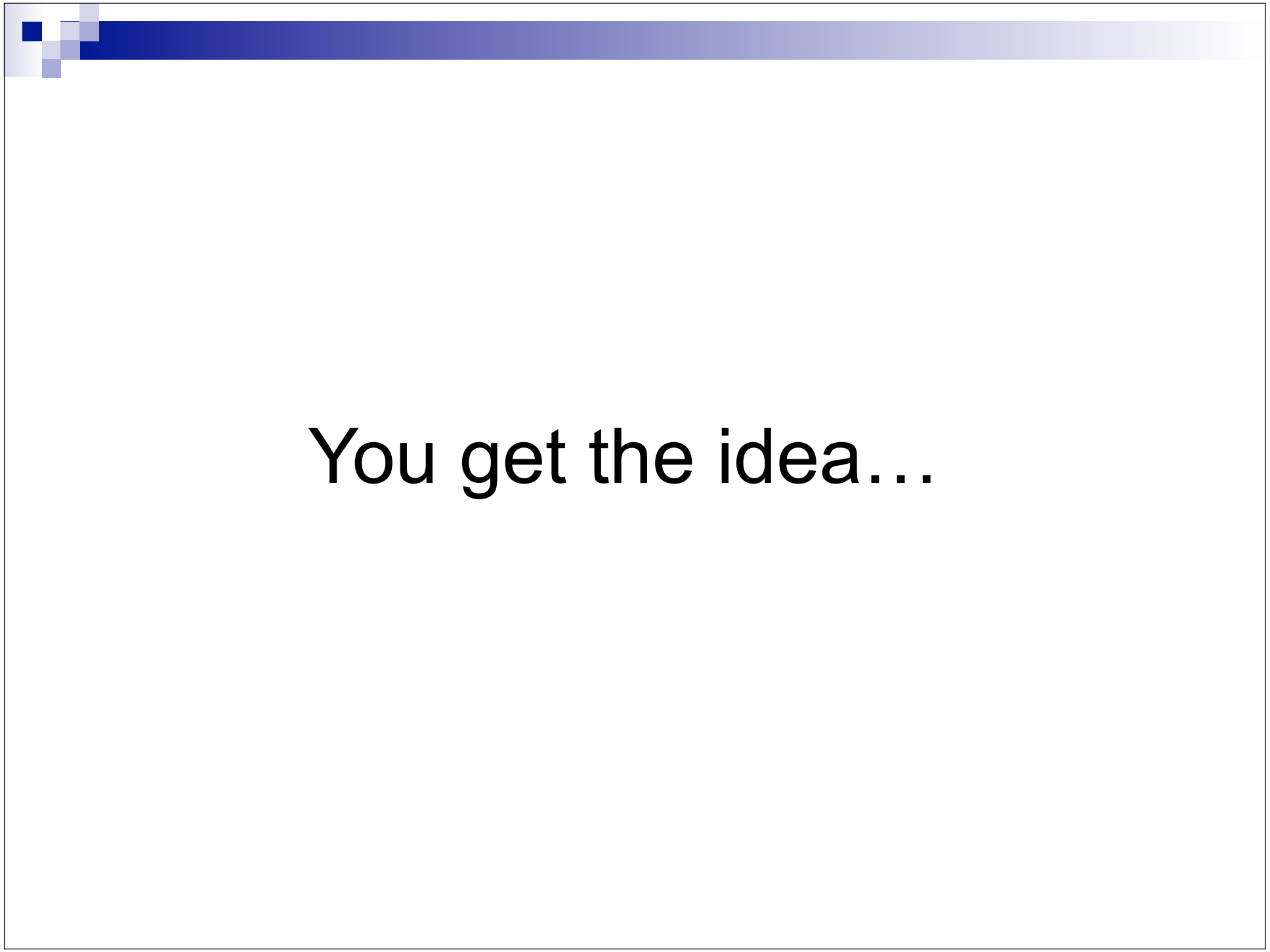
“\x01\x02\x03\x00\x59\x60\x4c\x4d\x4e”

PERMUTATION #5



STRUCTURE A (9 bytes)

“\x01\x02\x03\xFF\x59\x60\x4c\x4d\x4e”


You get the idea…

Maybe now you see Exponential Growth

- Depending on hierarchy of structures, the number of permutations grows (typically N*N)
- This is where mathematic Set Theory and Graph Theory assist us in intelligent data generation!!!!

RuXXers and "Set Theory"

- RuXXers generate mutations of primitives according to some defined logic
  - Length Calculator Off-By-One
  - Insertion of escape characters into String (SQL Injection)
- To be effective as a fuzzer all these mutations must be combined to generate all the possible mutations - this requires the application of mathematic "Set Theory"
- *In math terms RuXXers are "Set Morphisms" and can be bijective, injective or surjective functions*

RuXXers as Set Morphisms

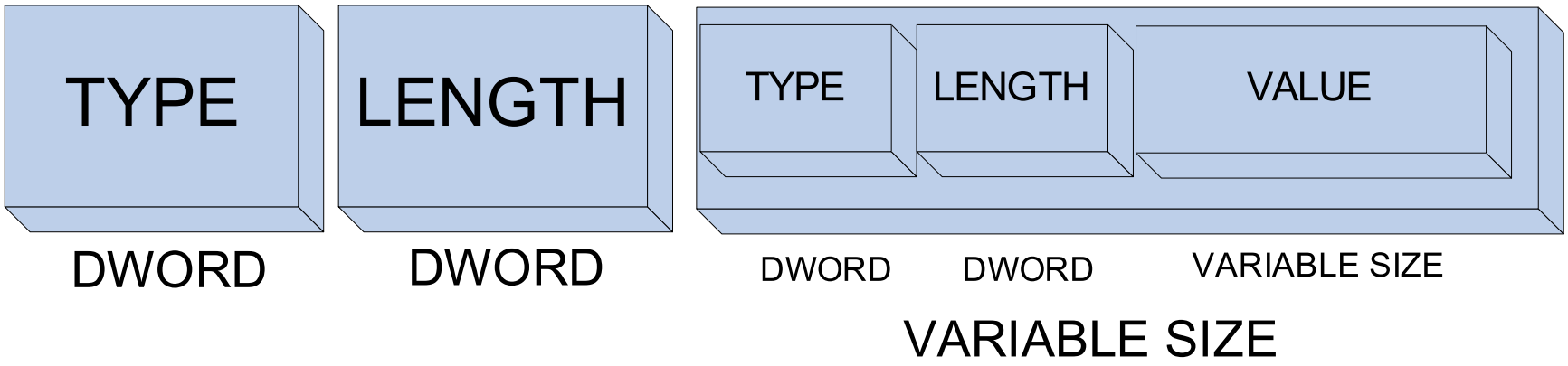
- With each Permutation we are actually calculating something called the *"Cartesian Product of N-Sets"* defined as:

$$X_1 \times \cdots \times X_n = \{(x_1, \ldots, x_n) \mid x_1 \in X_1 \text{ and } \cdots \text{ and } x_n \in X_n\}.$$

- RuXXer implements this to generate all possible permutations of Ruxxed Primitives

Example: A RuXXed TLV Protocol

- Let's return to our old hypothetical TLV protocol from earlier:



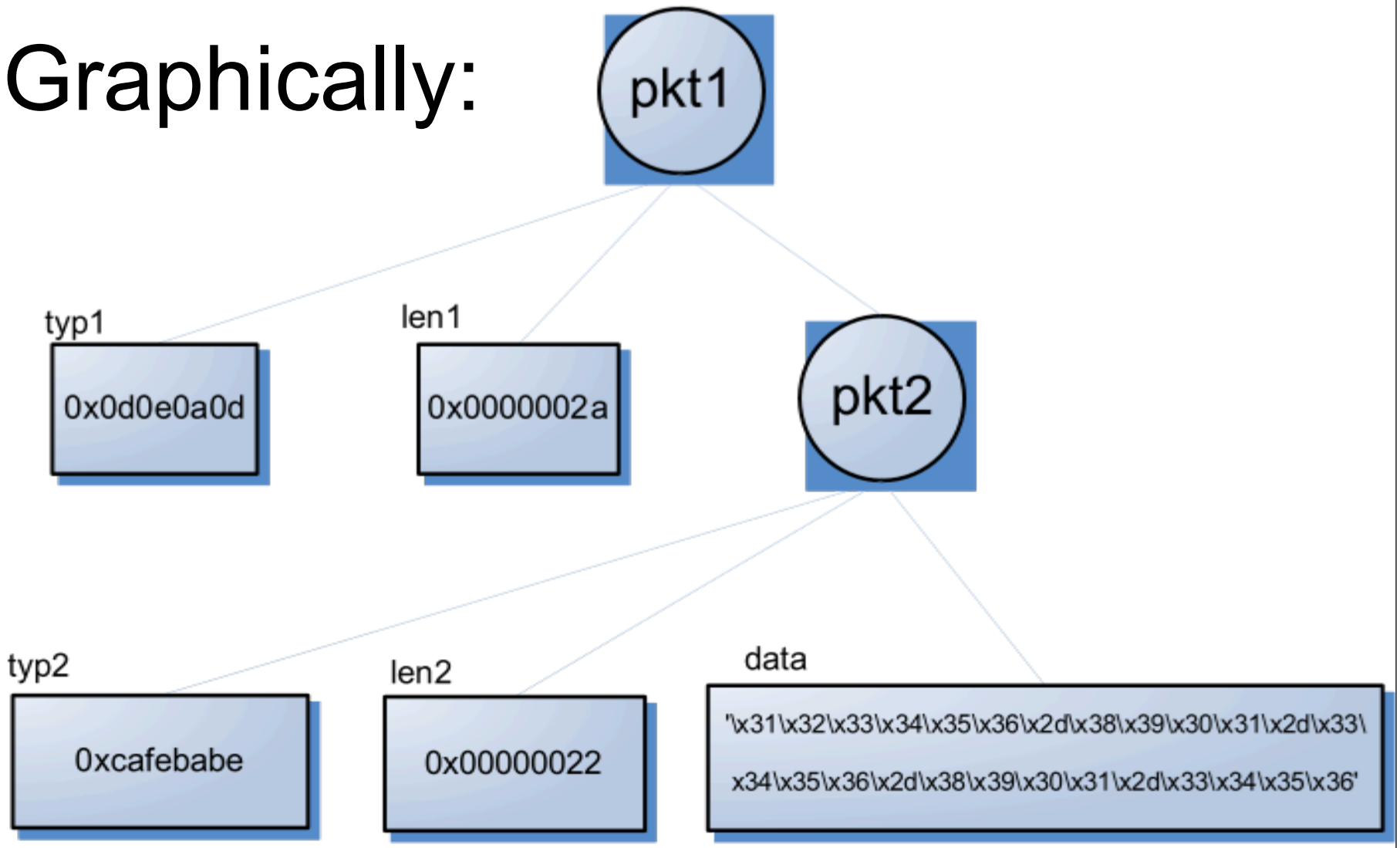
- We'll use "nested" TLV packets to further illustrate the point.

Our TLV Protocol in Ruxxer

```
structure pkt1, pkt2;
long typ1, typ2;
long_lc len1, len2;
string data;
typ1 = 0x0d030a0d;
typ2 = 0xcafebabe;
len1 = pkt1;
len2 = pkt2;
data = “123456-8901-3456-8901-3456”;
push(pkt2, typ2, len2, data);
push(pkt1, typ1, len1, pkt2);
```

- *# Ruxxer doesn’t actually support comma separated declarations yet. (done for space consideration)*

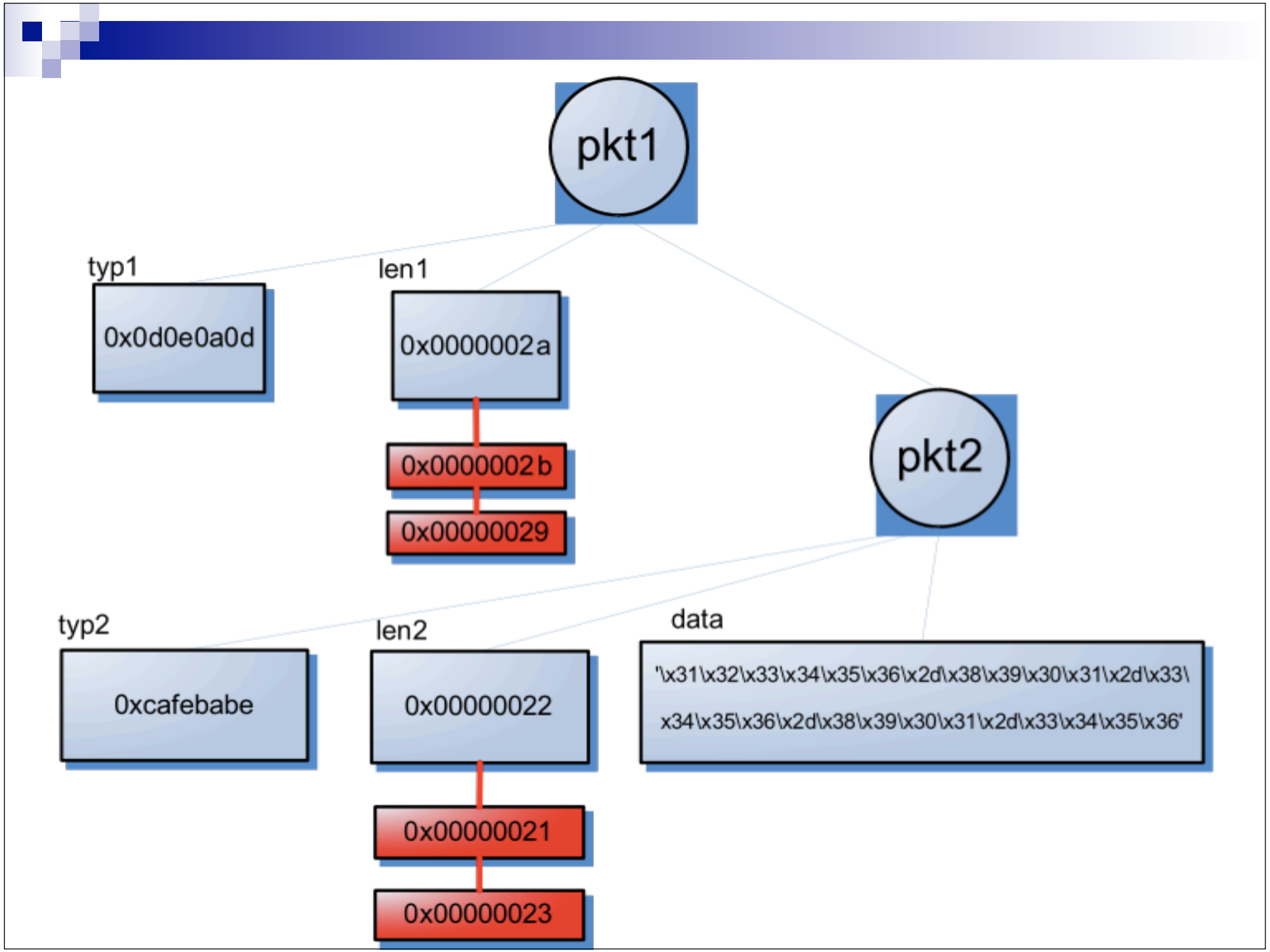
Graphically:



Our TLV Protocol in Ruxxer

```
structure pkt1, pkt2;
long typ1, typ2;
long_lc len1, len2;
string data;
typ1 = 0x0d030a0d;
typ2 = 0xcafebabe;
len1 = pkt1;
len2 = pkt2;
data = “123456-8901-3456-8901-3456”;
push(pkt2, typ2, len2, data);
push(pkt1, typ1, len1, pkt2);
OffByOne(pkt1);
ByteMinMax(pkt2);
```

- *# Ruxxer doesn’t actually support comma separated declarations yet. (done for space consideration)*


pkt1
typ1
0x0d0e0a0d
len1
0x0000002a
0x0000002b
0x00000029
pkt2
typ2
0xcafebabe
len2
0x00000022
0x00000021
0x00000023
data
'\x31\x32\x33\x34\x35\x36\x2d\x38\x39\x30\x31\x2d\x33\
x34\x35\x36\x2d\x38\x39\x30\x31\x2d\x33\x34\x35\x36'

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00\x2a\xca \xfe
\xba\xbe\x00\x00\x00\x22\x31\x32
\x33\x34\x35\x36\x2d\x38\x39\x30\x31
\x2d\x33\x34\x35\x36\x2d\x38\x39\x30
\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00\x2a\xca \xfe
\xba\xbe\x00\x00\x00**\x21**\x31\x32
\x33\x34\x35\x36\x2d\x38\x39\x30\x31
\x2d\x33\x34\x35\x36\x2d\x38\x39\x30
\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00\x2a\xca \xfe
\xba\xbe\x00\x00\x00**\x23**\x31\x32
\x33\x34\x35\x36\x2d\x38\x39\x30\x31
\x2d\x33\x34\x35\x36\x2d\x38\x39\x30
\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00**\x2b**\xca

\xfe\xba\xbe\x00\x00\x00**\x22**\x31\x32

\x33\x34\x35\x36\x2d\x38\x39\x30\x31

\x2d\x33\x34\x35\x36\x2d\x38\x39\x30

\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00\x2b\xca \xfe
\xba\xbe\x00\x00\x00**\x21**\x31\x32
\x33\x34\x35\x36\x2d\x38\x39\x30\x31
\x2d\x33\x34\x35\x36\x2d\x38\x39\x30
\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00\x2b\xca \xfe
\xba\xbe\x00\x00\x00**\x23**\x31\x32
\x33\x34\x35\x36\x2d\x38\x39\x30\x31
\x2d\x33\x34\x35\x36\x2d\x38\x39\x30
\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00**\x29**\xca

\xfe\xba\xbe\x00\x00\x00**\x22**\x31\x32

\x33\x34\x35\x36\x2d\x38\x39\x30\x31

\x2d\x33\x34\x35\x36\x2d\x38\x39\x30

\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00\x29\xca \xfe
\xba\xbe\x00\x00\x00**\x21**\x31\x32
\x33\x34\x35\x36\x2d\x38\x39\x30\x31
\x2d\x33\x34\x35\x36\x2d\x38\x39\x30
\x31\x2d\x33\x34\x35\x36”

9 Total Permutations of “pkt1”

“\x0d\x0e\x0a\x0d\x00\x00\x00\x29\xca \xfe
\xba\xbe\x00\x00\x00**\x23**\x31\x32
\x33\x34\x35\x36\x2d\x38\x39\x30\x31
\x2d\x33\x34\x35\x36\x2d\x38\x39\x30
\x31\x2d\x33\x34\x35\x36”

The "Resource Problem"

- Complex protocols with many nested structures can result in thousands of permutations
- Fuzzers that generate data have "resource problems" because they pre-expand all possible permutations…often exhausting memory
- RuXXer leverages the power of Python's ability dynamically manipulate/overload object attributes to completely eliminate this problem

Other RuXXer Features

- Fast-Forwarding to Iterations
- Various GUI Features
  - “Insert Bytes from File”
- Extensibility of Comms
- Extensibility of Language Interpreter

Conclusion

- Dumb, non-protocol aware fuzzing is not sufficient
- Existing fuzzing frameworks sacrifice easy usability for power, or vice-versa.
- RuXXer achieves a balance by placing a simple language on top of a powerful fuzzing framework

http://www.ruxxer.org

- Download RuXXer Bundle (or source)
- Get RuXXer Updates
- Read RuXXer Wiki
- Browse RuXXer SVN Repository
- Submit Bugs/Feature Requests/Ideas/ Brainstorms

Questions / Comments

- Email:
  - stephen@ruxxer.org
  - colin@ruxxer.org