



# fuzzing フレームワー ク fuzzing 言語?!

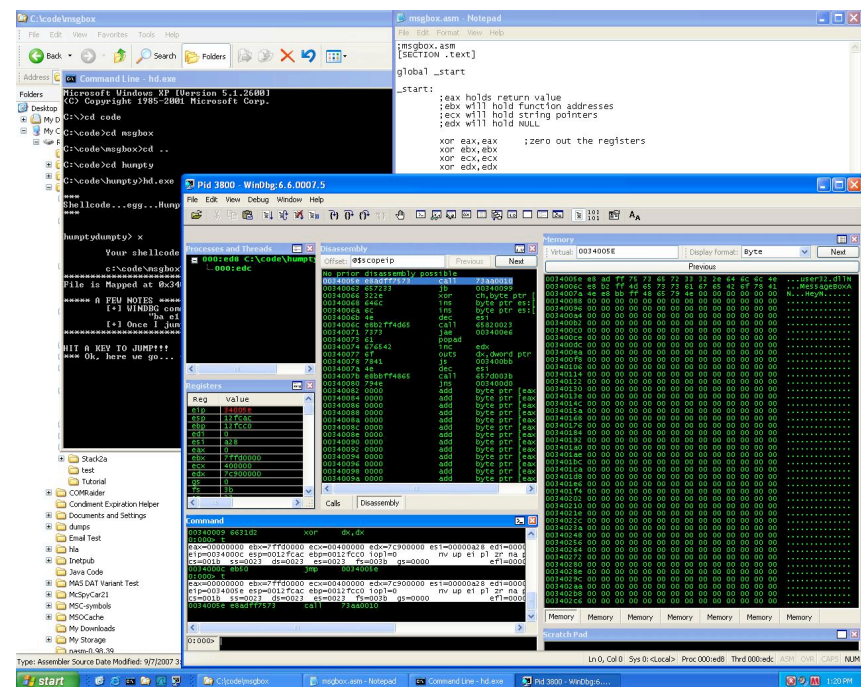
**Stephen “sa7ori” Ridley, McAfee シニア セキュリティ アーキテクト  
(元)**

(DoD -> McAfee -> Independent RCE/Vuln Researcher)  
stephen@blackroses.com

**Colin Delaney, McAfee ソフトウェア セキュリティ エンジニア**

# 「Fuzzing」とは何か？

- セキュリティの欠陥を見つけることを目的とした対象アプリケーションのストレステスト
- 破壊されたデータを対象アプリケーションに提供し、パーサとデータ欠陥口ジックにストレスを与える



# ブラインド / ダム fuzzing

- 「プロトコル対応」ではない
- ランダム データによりインプットのランダム セクションを破壊
- テストの幅と深さを最小化
- 手早く容易に使用できるが、「シャドー」バグを

```
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";
```

# スマート (プロトコル対応) fuzzing

- インプット プロトコル  
は、fuzzingの試みをサ  
ポートするために複製さ  
れる
- fuzzerは、データ タイプ  
に対応しており、インテ  
リジェントな反復を提供  
可能
- データ タイプに基づき、

```
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";
```



## 「スマート fuzzing」の問題点

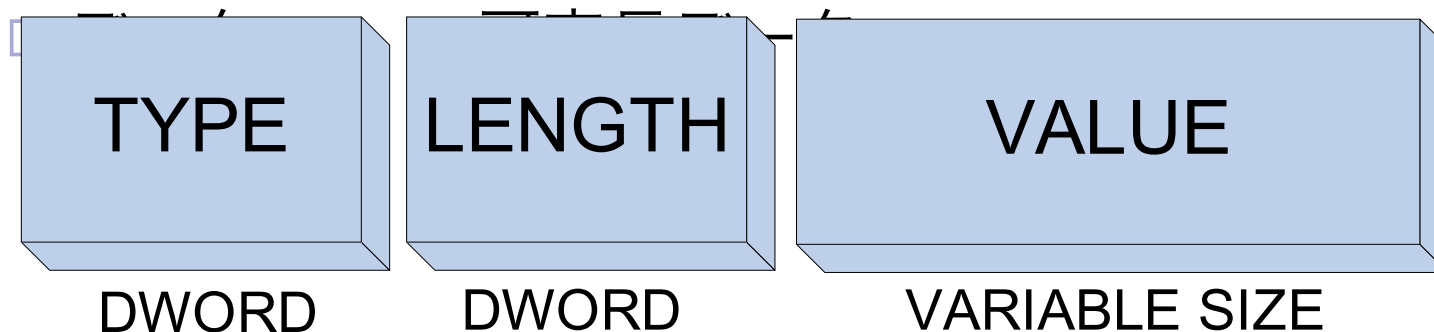
- プロトコルの手作業での複製は高額のコストがかかる
- スマート fuzzingはターゲットになる可能性が高く、コードの再利用は容易でない
- ある fuzzingの改善または技術革新は、何らかのフレームワークまたはオブジェクトモデルがない場合には、他の fuzzing プロ

# ダム & スマートfuzzing

## 対 仮説プロトコル

### ■ 例: タイプ-長さ-値 (TLV) プロトコル

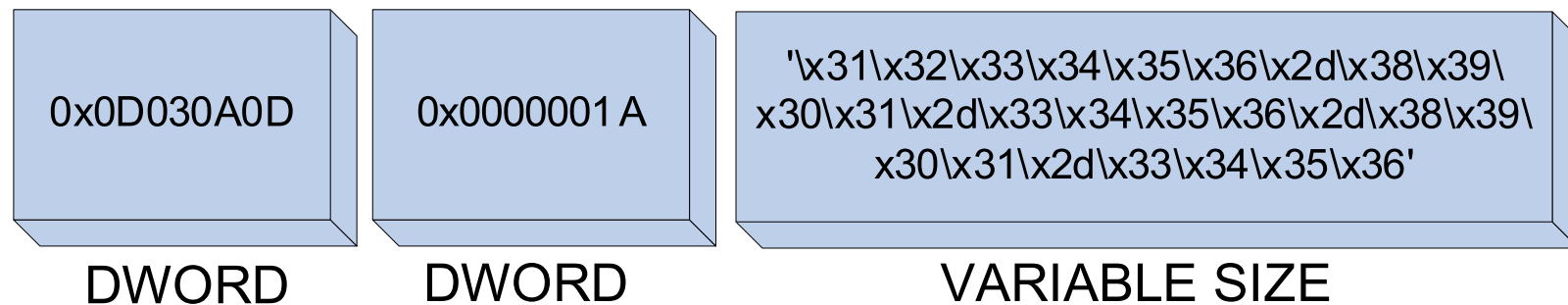
- タイプ:           タイプ フィールド
- 長さ:   次のセクションの長さ



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

# 例: ダム / ブラインド fuzzing

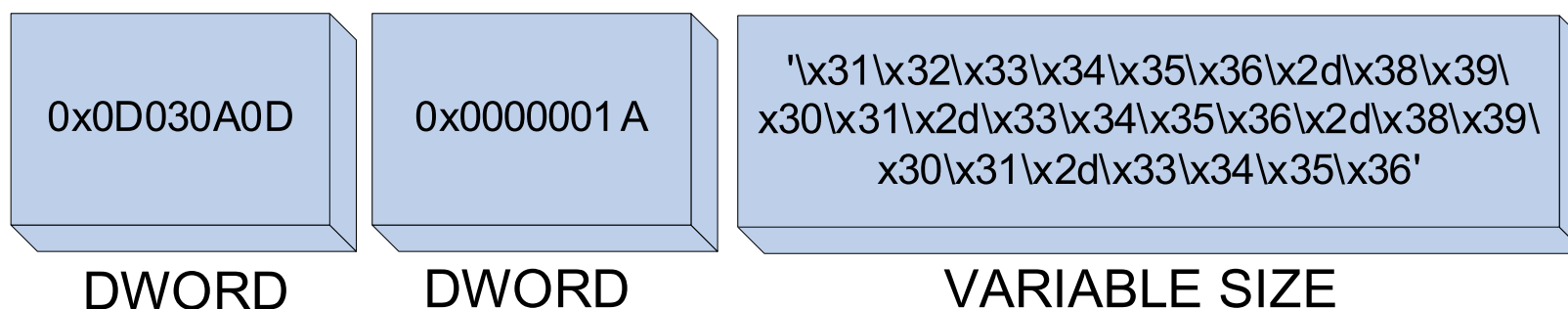
- ブラインドのfuzzerは、このブロックを移動し、バイト数や文字などをランダムに変更する
- 大半のランダムな反復は、プロトコルに準拠していないため、早期に破棄される



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

# 例: スマートfuzzing

- インテリジェントな fuzzer とは
  - プロトコルに対応している
  - すべてのフィールドに有意の反復
  - Fuzzingしないフィールドを保持
  - ダイナミックに長さを計算できる



0D030A0D	0000001A	FFFFFF-FFFF-FFFF-FFFF-FFFF
0D030A0D	00000001	0
0D030A0D	FFFFFFFF	AAAAAAAAA [...] AAAAAAAAAAAAAAAAAA





# 最新のfuzzingソリューションの検証

- 現在利用可能なfuzzerには多くのものがあるが、今回の検証は、次の主要なfuzzerに限定する:

- ☐ SPIKE
- ☐ Sully
- ☐ Peach

# 最新のfuzzerの検証:

## SPIKE

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

### ■ 利点

- 幅広く利用されている
- 強力

### ■ Ruxxerの利点

- WindowsおよびLinuxで動作
- 「C」コーディングの知識は不要
- オブジェクトモデルは、fuzzingプロトコルで容易に

# 最新のfuzzerの検証:

## Sully

### ■ 利点

- API / フレームワーク ベース
- デバッガ、ターゲット モニタ
- コード カバレッジ指標



### ■ Ruxxerの利点

- Pythonのインストールが不要なスタンドアローンEXE
- 不慣れなユーザ向けの、抽象化されたシンプルなスクリプトにより、利用しやすくなっている
- エキスパートはスクリプト言語を無視して、「API

# 最新のfuzzerの検証:

## Peach

### ■ 利点

- API / フレームワーク
- 拡張可能

### ■ Ruxxerの利点

- APIはもともとユーザフレンドリーではないが、フレームワークの上部にグラフィカルなIntegrated Development Environment (統合型開発環境、IDE) を追加することで、セキュリティ以外の分野のエンジニアにも容易に消化可能なものとなっている

#### Submodules

- [Peach.Generators](#): Default included Generators.
  - [Peach.Generators.block](#): Contains implementation of block generators.
  - [Peach.Generators.data](#): Common data generators.
  - [Peach.Generators.dictionary](#): Contains generators that use dictionaries.
  - [Peach.Generators.flipper](#): Default flippers.
  - [Peach.Generators.incrementor](#): Incrementing generators.
  - [Peach.Generators.null](#): These Generators evaluate to null.
  - [Peach.Generators.repeater](#): Generators that repeat a value.
  - [Peach.Generators.static](#): Default static generators.



# fuzzingの世界における変化 (fuzzingの革命)

1. ブラインドfuzzer
2. ほどよくプロトコルに対応したfuzzer
3. プロトコルに対応したfuzzingフレームワーク
4. 基本データ変異機能を持つfuzzingフレームワーク
5. データ変異とビジュアル化機能を備えたフレームワーク
6. データのビジュアル化機能、高機能データ変異機能を備えた完全なfuzzing言語



# 新しいアプローチの論理的根拠

- fuzzingは進化を続けている:
  - fuzzingは現在、品質保証 (QA) 分野でプレリリーステスト活動として利用されている
  - エンジニアリング部門は、セキュリティ開発ライフサイクル (SDL) の側面を、ソフトウェア開発ライフサイクル (SDLC) に組み込む作業を続けている
  - 最新の状況を通じていることを追求するセキュリティの専門家は、「骨の折れる」 fuzzingに対処するための革新的なツールを必要としている



## ... 新しいアプローチ (続き)

- 限定的なセキュリティ知識しか持たないエンジニアは、乏しい予算内で、自社のソフトウェア スイートのfuzzingを始める必要がある  
... しかし ...
- セキュリティの専門家は、使い勝手のた




## ... 新しいアプローチ: 妥協点!

- API / フレームワークは非常に強力だが、学習曲線面の理由から、即効性には乏しい
- 妥協点は、以下のようなシンプルなスクリプト言語を通じてフレームワークを利用できるようにすることである:



ついに登場

**RuXXer**  
The Fuzzing Language



# RuXXerアーキテクチャ

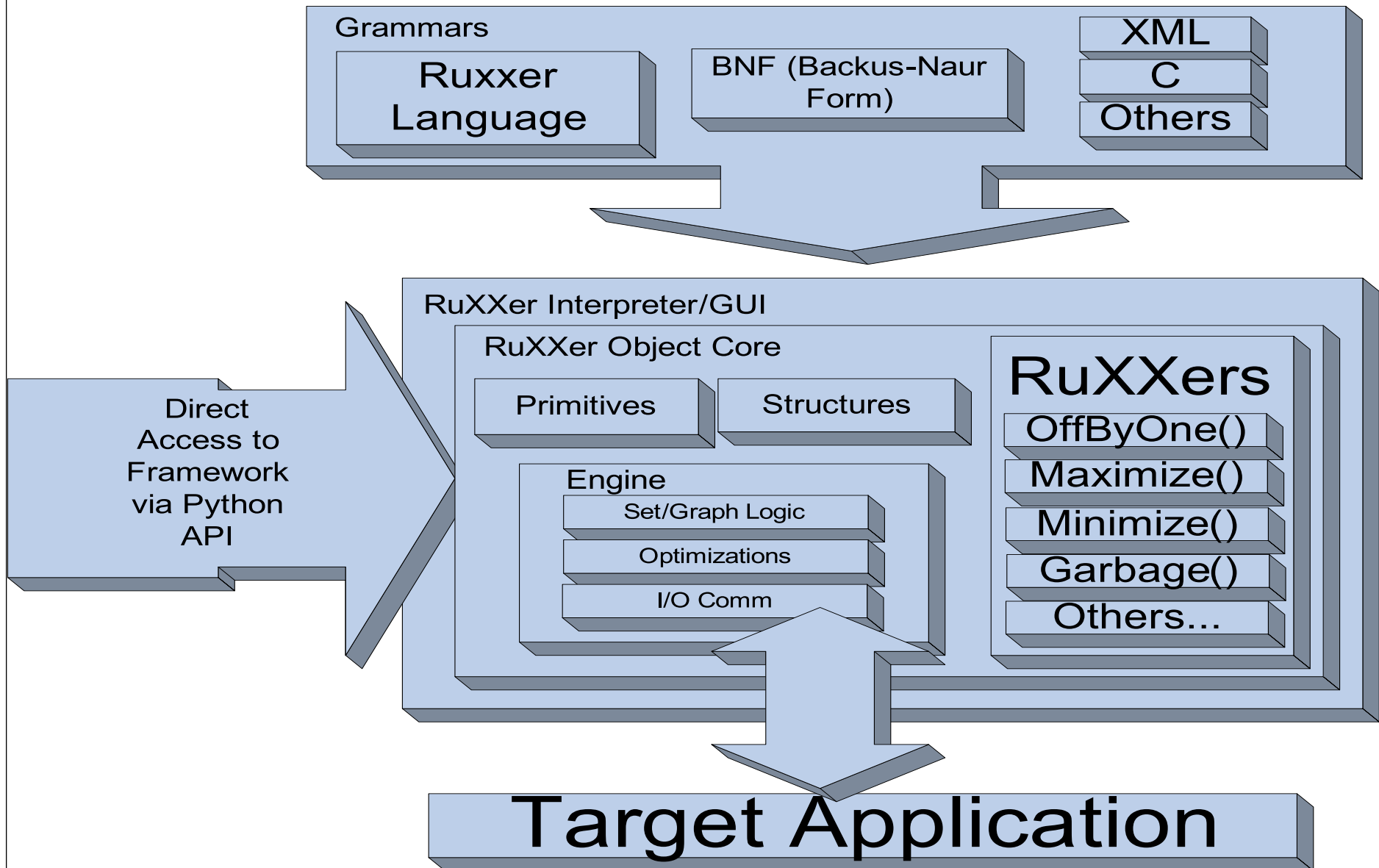
- 簡単に言えば: RuXXerは、独自の言語を備えた、強力なfuzzingフレームワークである
- 2つの主要な部分から構成される:
  - 言語インタープリタ
  - fuzzingフレームワーク コア
    - オブジェクト モデル

# デモ

(必要な機能にすべて対応)

- なかなか良さそうだが、外観や使用感はどうか？
- fuzzingフレームワークは、ライブラリを備えたAPIだが、「fuzzing言語」にはどのような機能があるのか？
- RuXXer アプリケーション ウィンドウ:
  - コーディング ウィンドウ

# RuXXerアーキテクチャ





# Ruxxer オブジェクト コア


- 直感的なC風の言語
- (オブジェクト モデルであるため) Pythonから直接API/フレームワークとして使用可能
- コアは、次の4つの概念を元に構築されている:
  - プリミティブ
  - 構造
  - RuXXer
  - Comm

# オブジェクト コア: プリミティブ

- データの最も基本的な形式:
  - Sulleyの「レゴ」のようなもの
- 基本的なRuXXerプリミティブ:
  - バイト
  - ショート
  - ロング
  - 文字列
  - 長さ計測機能

# オブジェクト コア: 構造

- プリミティブのコンテナ
  - Sulleyの「ブロック」のようなもの
- 抽象データ タイプの基本的な構築ブロック
- C「構造」に論理的に近似
  - ただし不透明性は低い
  - 実際のインスタンス



# オブジェクト コア: Comm

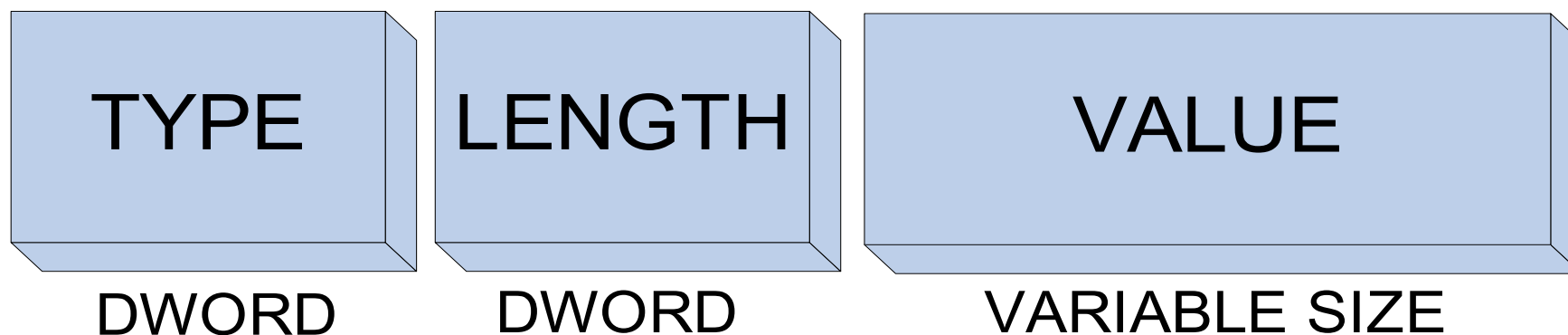
- ターゲットにデータを提供
  - ☐ TCPClient
  - ☐ TCPServer
  - ☐ UDPClient
  - ☐ UDPServer
  - ☐ FileOutput
- 拡張性が高い、追加も容易...



# RuXXerの例: TLVプロトコル

- 対象: シンプルなTLV (タイプ-長さ-値プロトコル) をベースにした仮説プロトコル

□ RIFF、PNGなど



# RuxxerにおけるTLV プロトコルのモデ

## リング

#declarations

long typeop;

long\_lc pkt\_len; #byte length calculator

string dgram;

structure tlv\_packet;

#assignments

typeop = 0x0D030A0D;

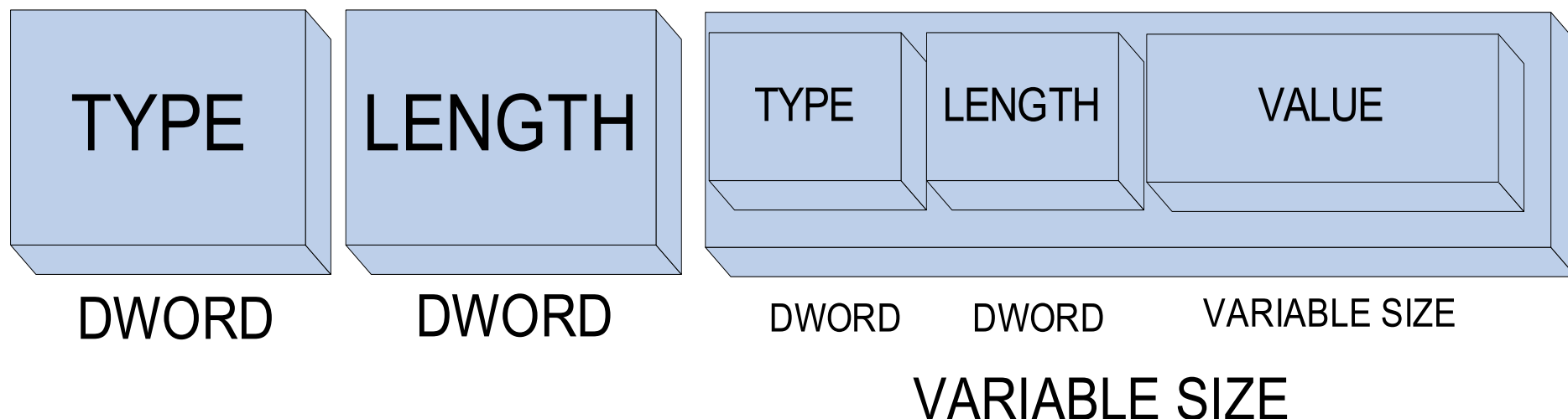
pkt\_len = tlv\_packet;

dgram = "This is userdata\r\nm"

push(tlv\_packet, typeop, pkt\_len, dgram);

# ネストされたプロトコル

- RuXXerの「構造(Structure)」タイプは、ネストされた複雑なデータ構造を示すよう設計されている





#declarations

long typeop;

long\_lc pkt\_len; #byte length calculator

structure dgram; # push(dgram, ..., ..., ...

structure tlv\_packet;

#assignments

typeop = 0x0D030A0D;

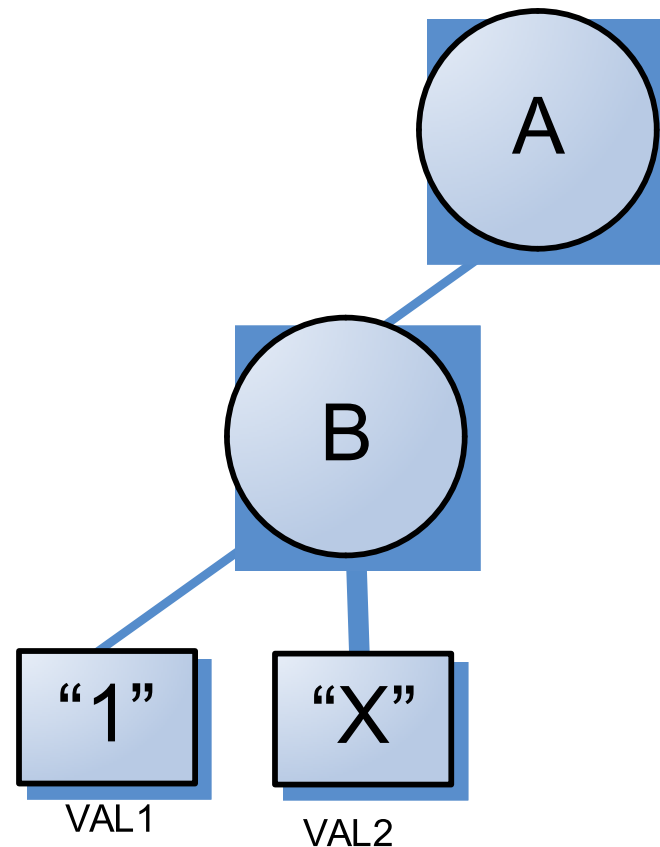
pkt\_len = tlv\_packet;

dgram = "This is userdata\r\n"

push(tlv\_packet, typeop, pkt\_len, dgram);

# グラフィカルな表示

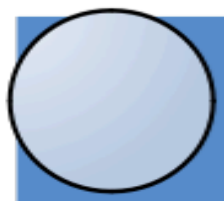
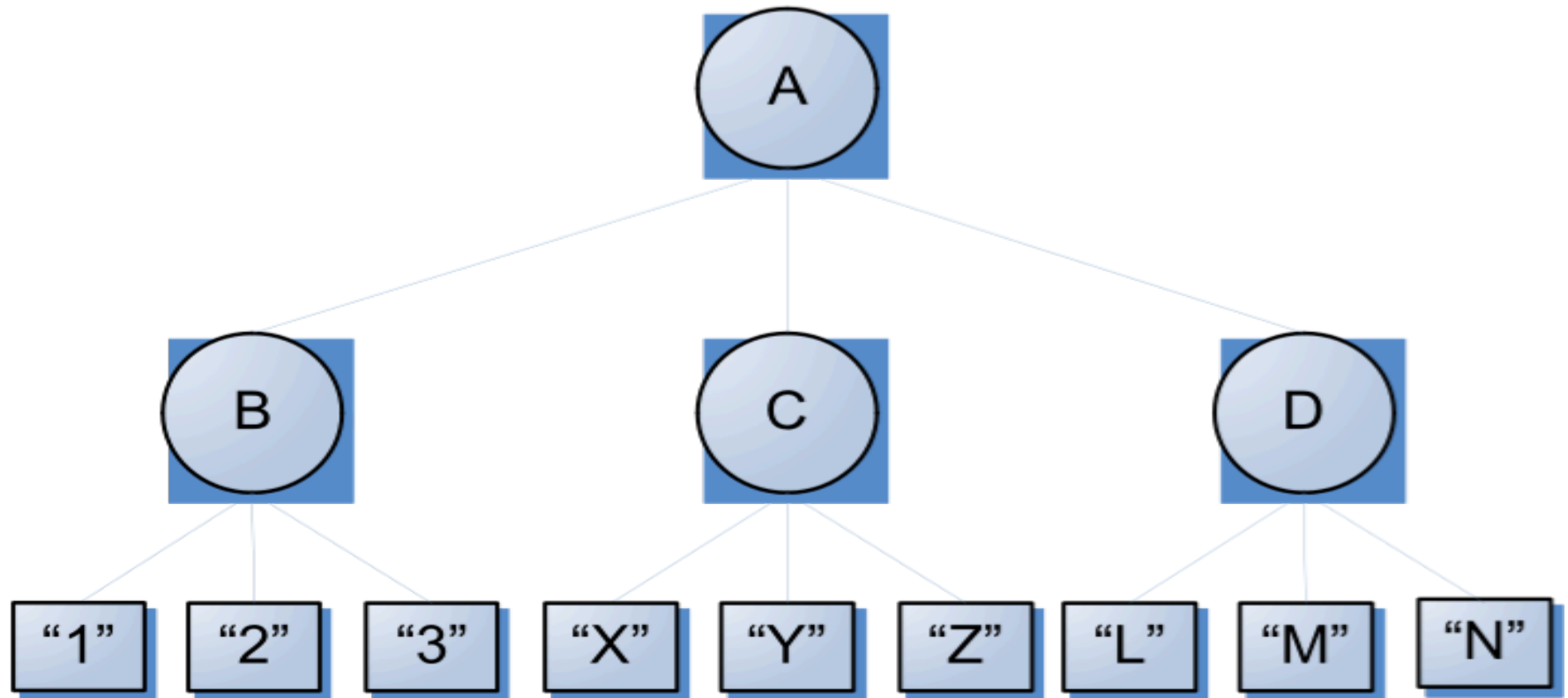
```
structure A;  
structure B;  
int val1;  
val1 = 1;  
string val2;  
val2 = "X";  
#now we push  
push(B, val1, val2);  
push(A, B);
```



# RuXXer: インテリジェントなデータ変換

## ■ RuXXer

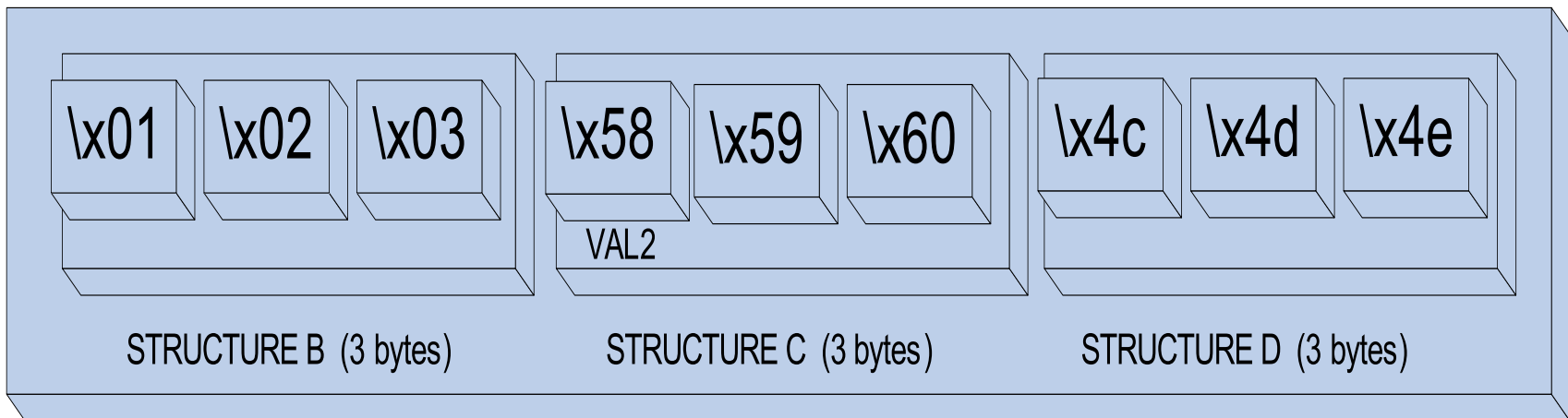
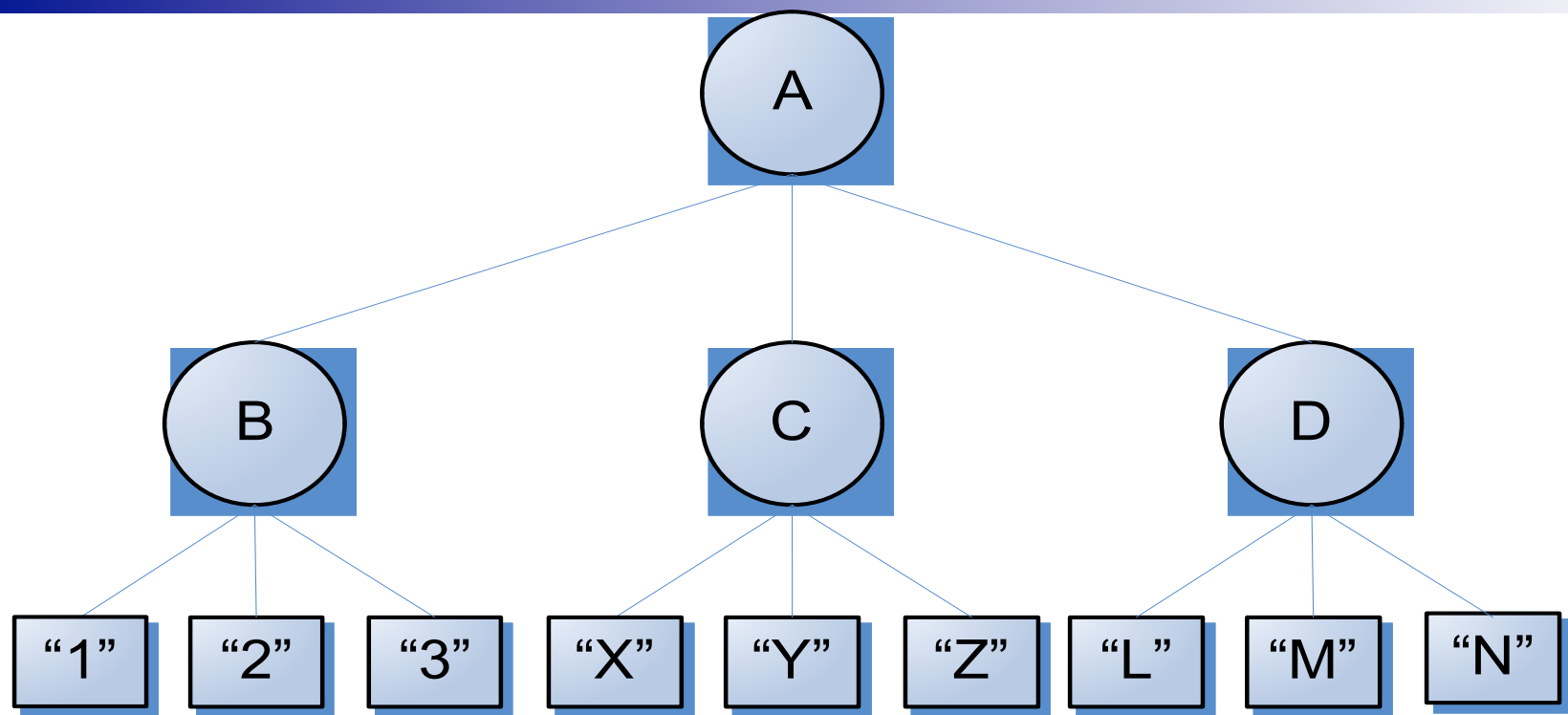
- 指定された方法でプリミティブの値を変化
- 「再利用可能なテスト ケース」 の概念の基礎を形成
- 特定のプリミティブ タイプに適用され、ポイントレスなテスト ケースを解消
  - 例: 数値フィールドで実行する文字列テスト
- 拡張性が容易、常に機能追加が可能



= STRUCTURE

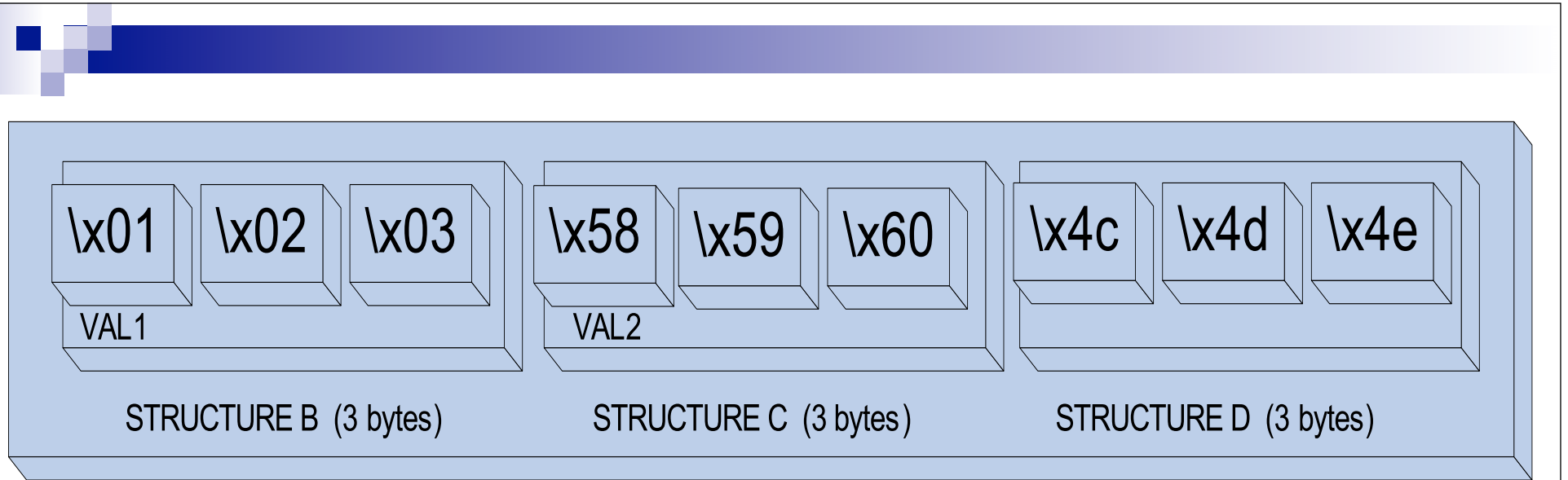


= PRIMITIVE



STRUCTURE A (9 bytes)





STRUCTURE A (9 bytes)

```
print(A);
```

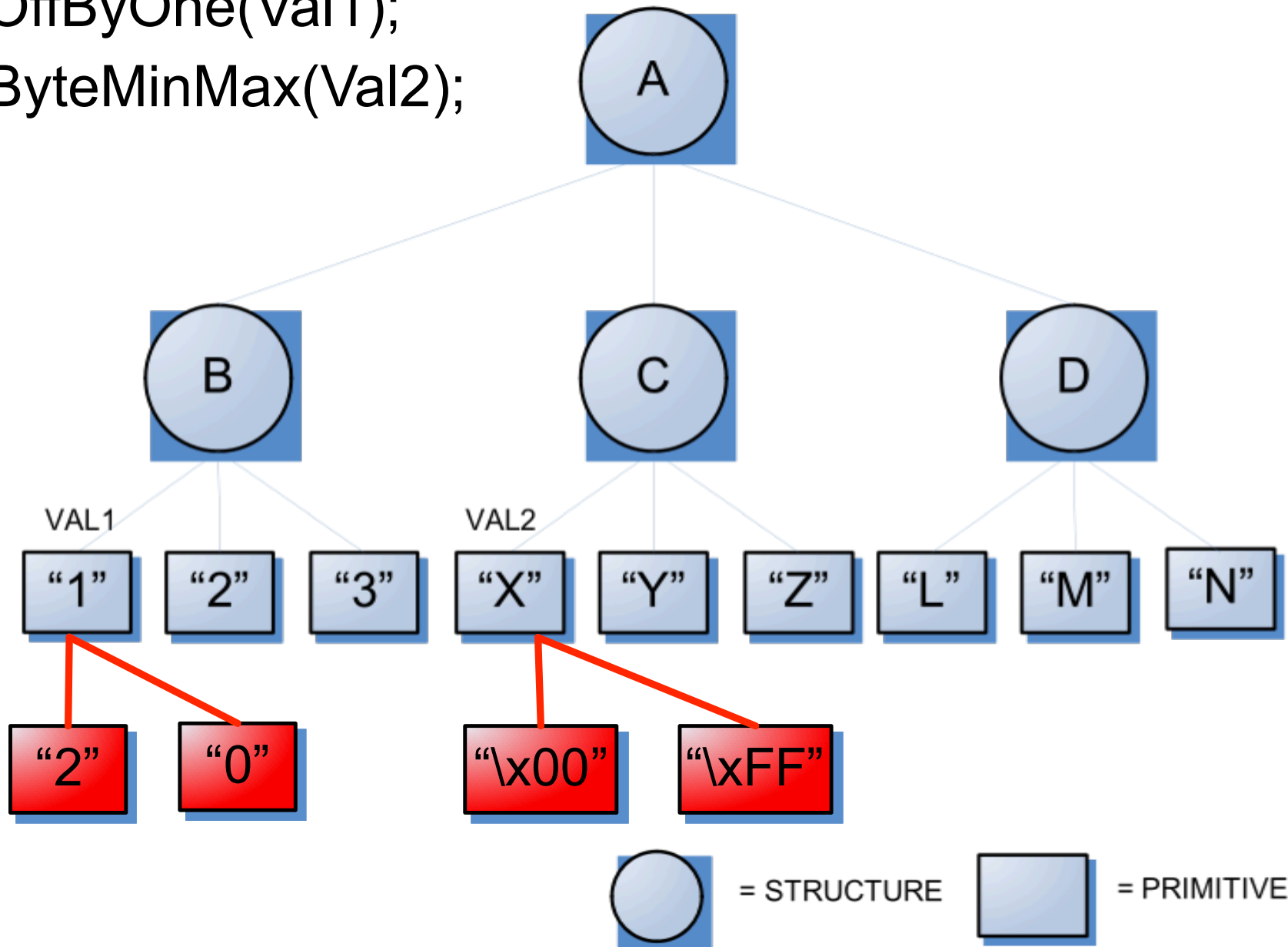
```
>> "\x01\x02\x03\x58\x59\x60\x4c\x4d\x4e"
```

```
print(B);
```

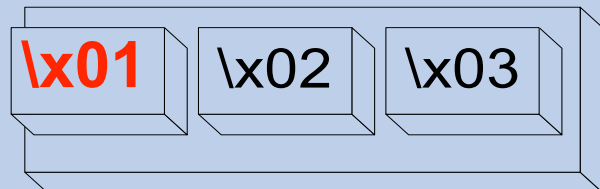
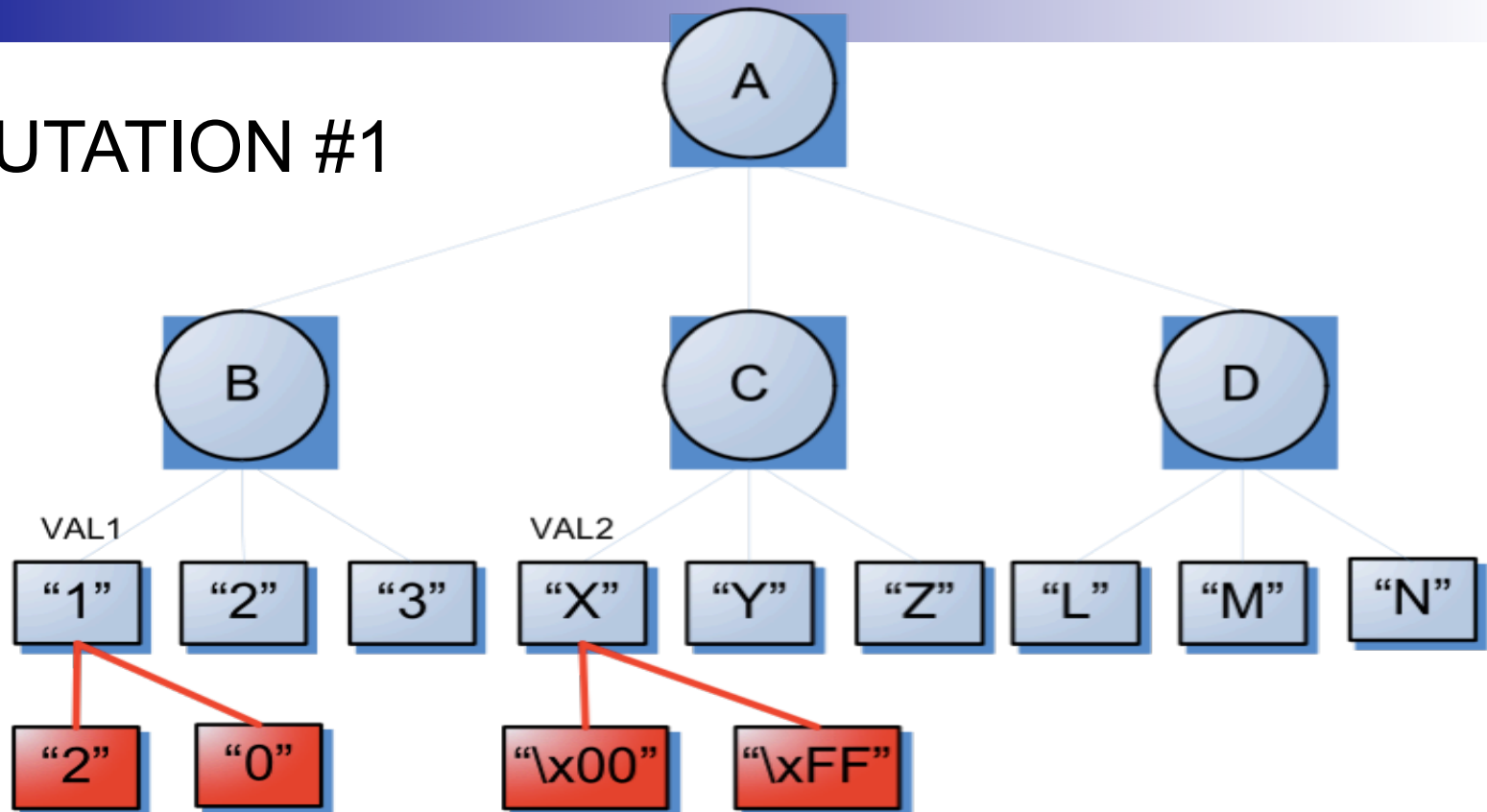
```
>> "\x58\x59\x60"
```

```
.....
```

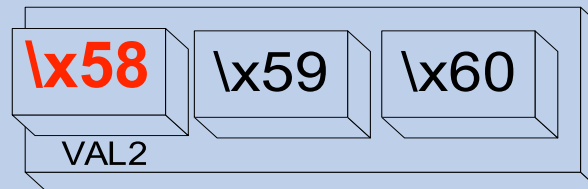
OffByOne(Val1);  
ByteMinMax(Val2);



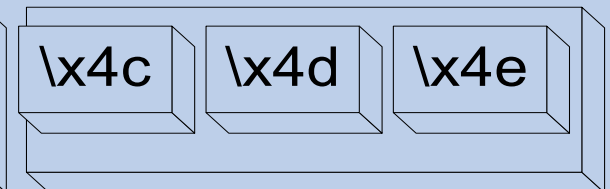
# PERMUTATION #1



STRUCTURE B (3 bytes)



STRUCTURE C (3 bytes)

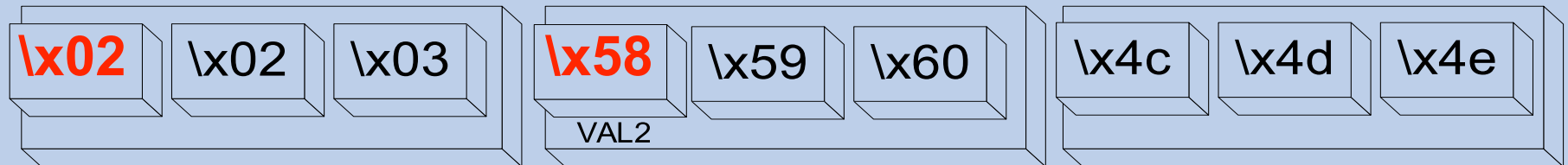
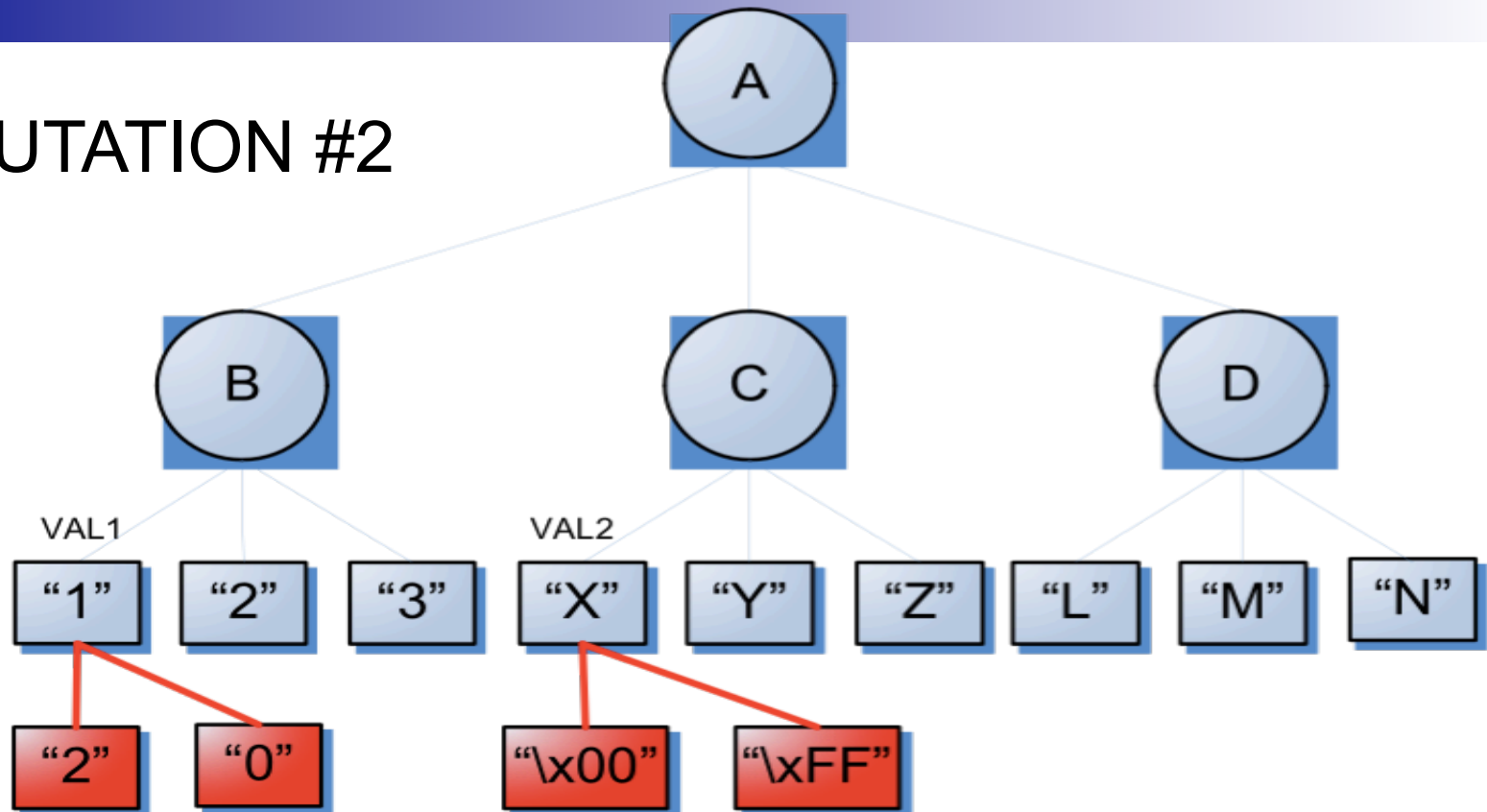


STRUCTURE D (3 bytes)

STRUCTURE A (9 bytes)

"\x01\x02\x03\x58\x59\x60\x4c\x4d\x4e"

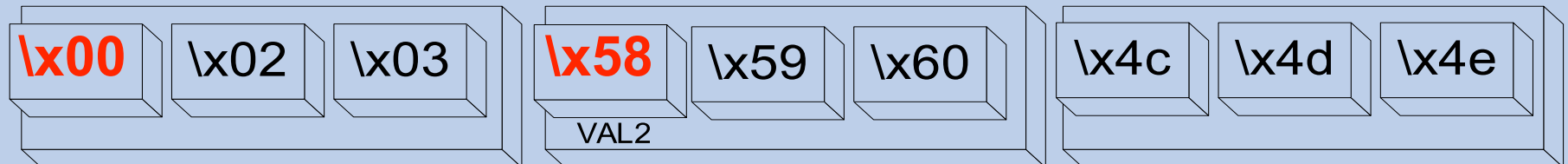
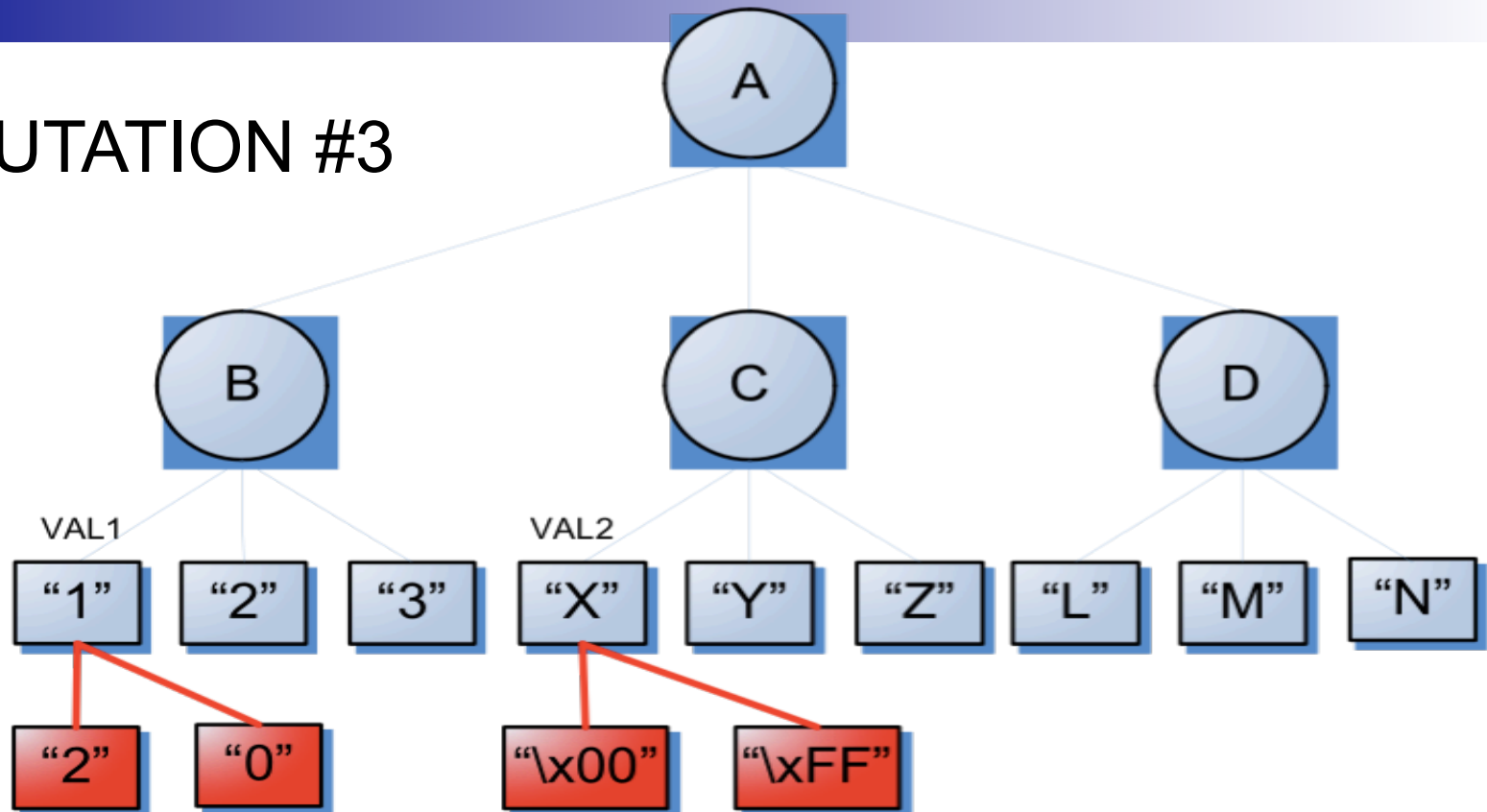
## PERMUTATION #2



STRUCTURE A (9 bytes)

"\x02\x02\x03\x58\x59\x60\x4c\x4d\x4e"

## PERMUTATION #3



STRUCTURE B (3 bytes)

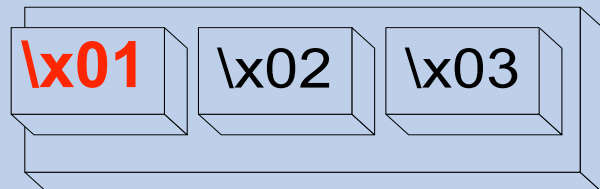
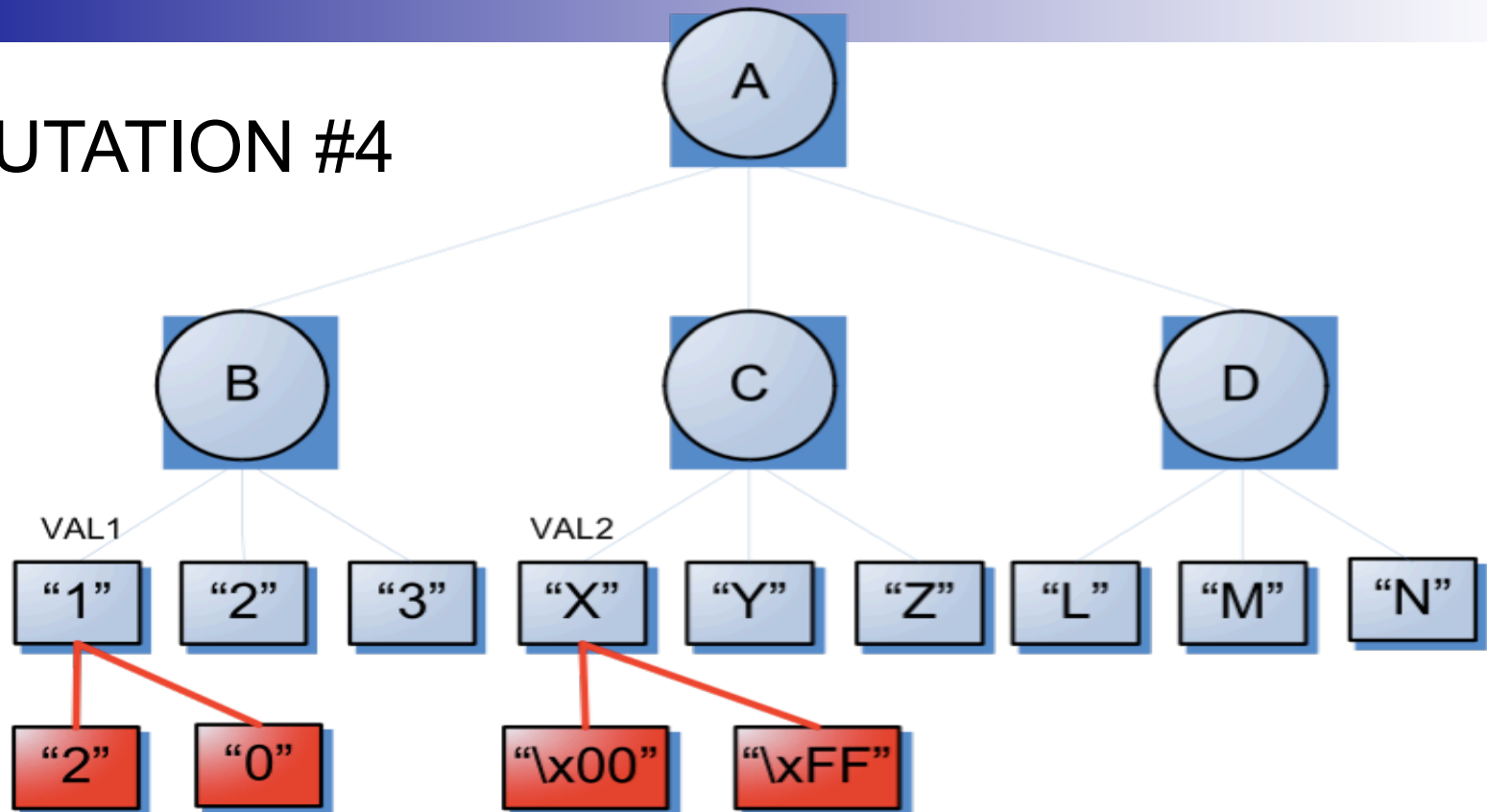
STRUCTURE C (3 bytes)

STRUCTURE D (3 bytes)

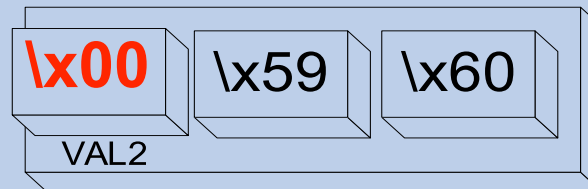
STRUCTURE A (9 bytes)

"\x00\x02\x03\x58\x59\x60\x4c\x4d\x4e"

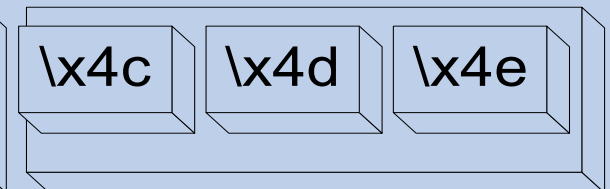
## PERMUTATION #4



STRUCTURE B (3 bytes)



STRUCTURE C (3 bytes)

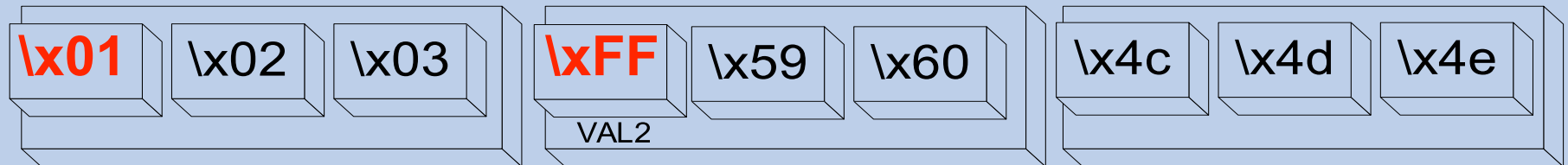
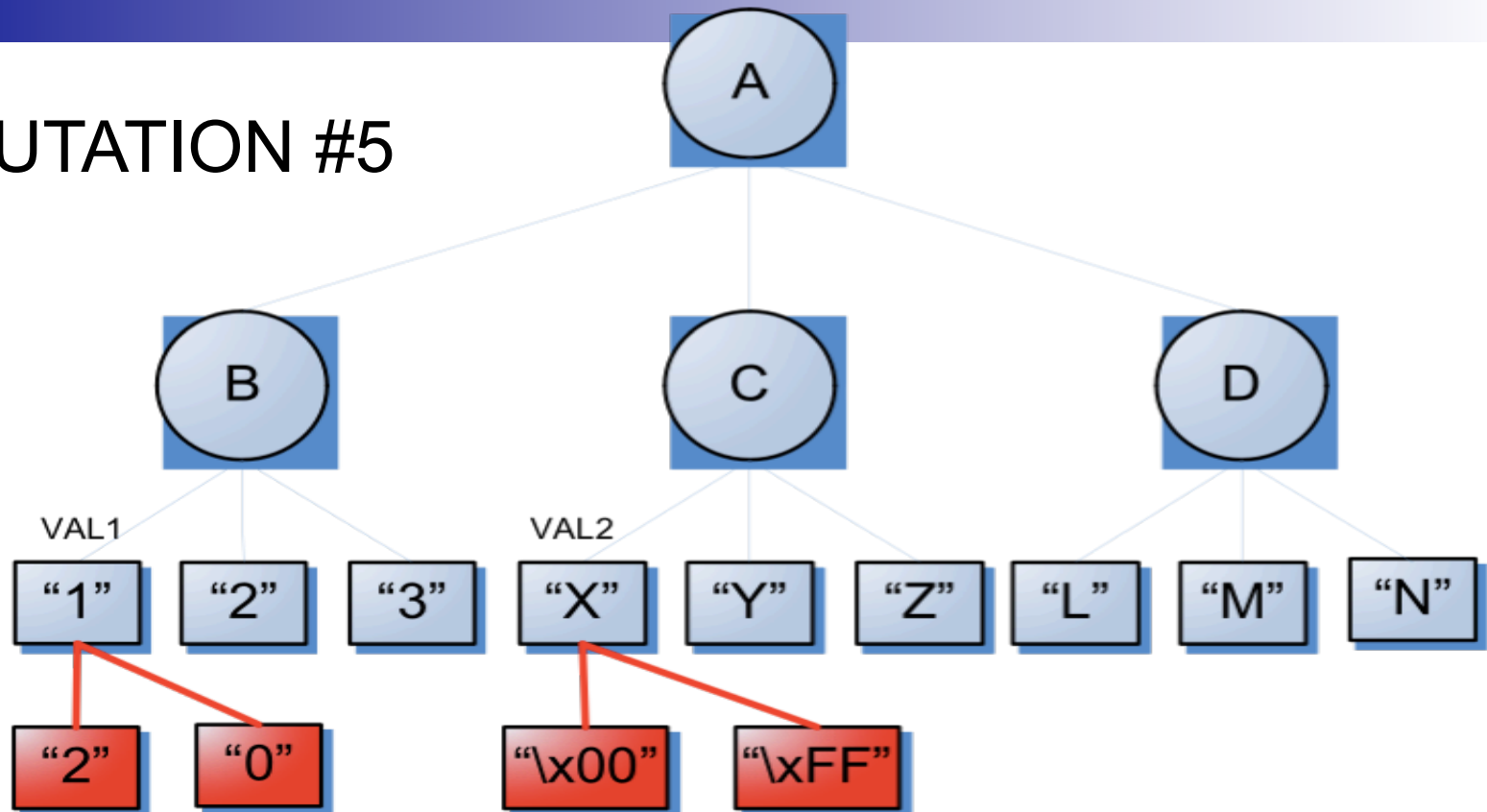


STRUCTURE D (3 bytes)

STRUCTURE A (9 bytes)

"\x01\x02\x03\x00\x59\x60\x4c\x4d\x4e"

## PERMUTATION #5



STRUCTURE B (3 bytes)

STRUCTURE C (3 bytes)

STRUCTURE D (3 bytes)

STRUCTURE A (9 bytes)

"\x01\x02\x03\xff\x59\x60\x4c\x4d\x4e"



アイディアが生まれる...





## 恐らくあなたは**急激な成長**を 目の当たりにすることでしょう

- 構造の階層に基づき、順列の数値は大きくなっていく (通常は $N*N$ )
- 数学的な Set Theory (集合論) と Graph Theory (グラフ理論) の援用により、インテリジェントなデータ生成が行える!!!!



# RuXXer と 「集合論」

- RuXXerは、定義済みのロジックに従って、プリミティブの変化を生成します
  - 長さ計算機能 Off-By-One
  - 文字列へのエスケープ文字の挿入 (SQL インジェクト)
- fuzzerとして効果を発揮するため、可能性のあるすべての変化を生成するため、これらすべての変化を結合する。これには、数学的「集合論」の応用が必要となる
- 数学的な観点から、*RuXXer*は、「Set Morphisms」であ

# Set Morphism としての

## RuXXer

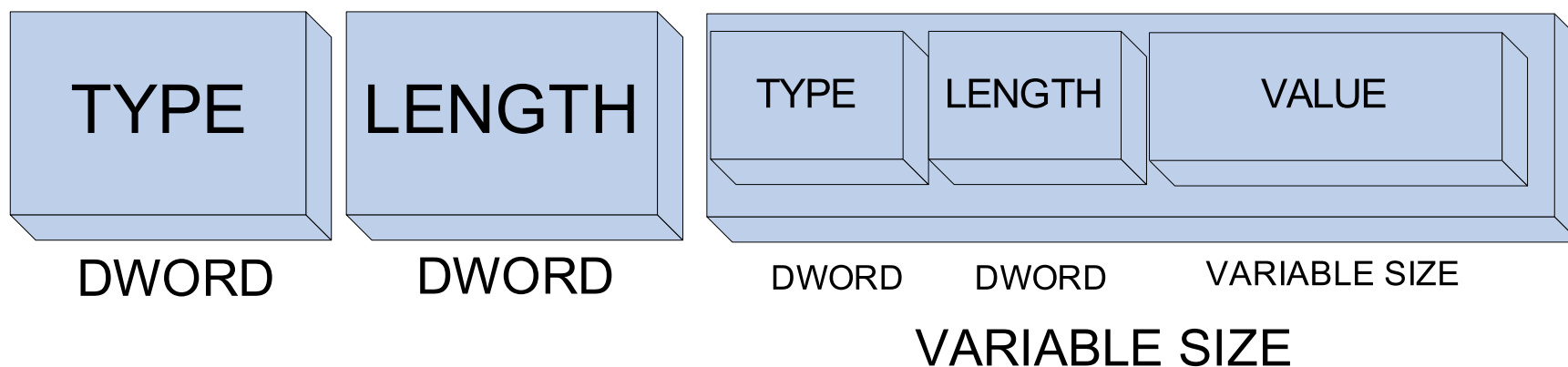
- 各順列を利用して、次のように定義された「*Cartesian Product of N-Sets*」と呼ばれるものを実際に計算してみる:

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

- RuXXerは、これを実行して、RuXXer化されたプリミティブのすべての順列を生成する

## 例: RuXXer化されたTLVプロトコル

- 前半で述べた古い仮説TLVプロトコルに立ち返ってみたい:



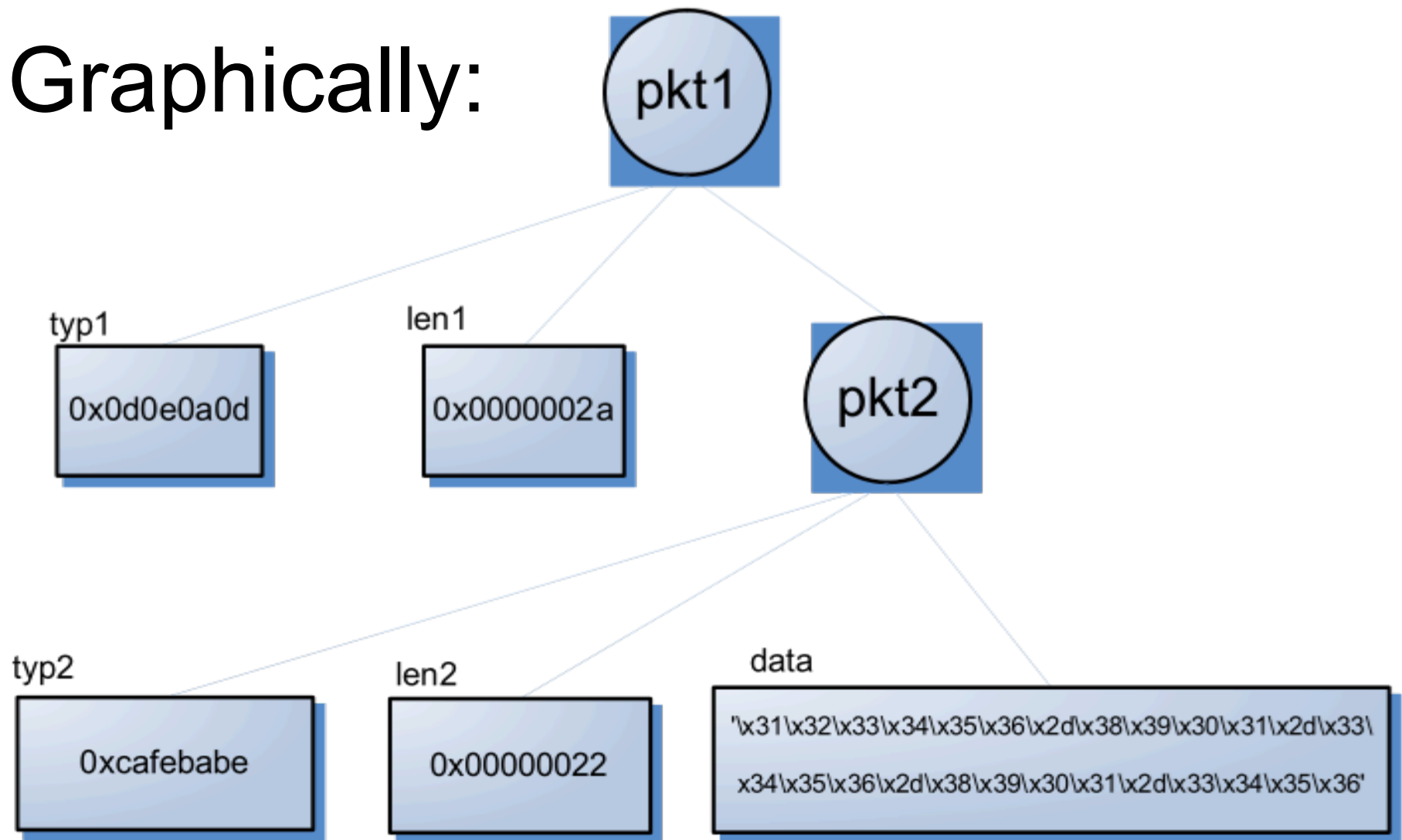
- より詳しく見ていくため、「ネストされた」TLV パケットを使用する。

# RuxxerにおけるTLVプロトコル

```
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*は、実際にはカンマ区切りをサポートしていない (ここではスペースを意味するためにカンマを使用している)

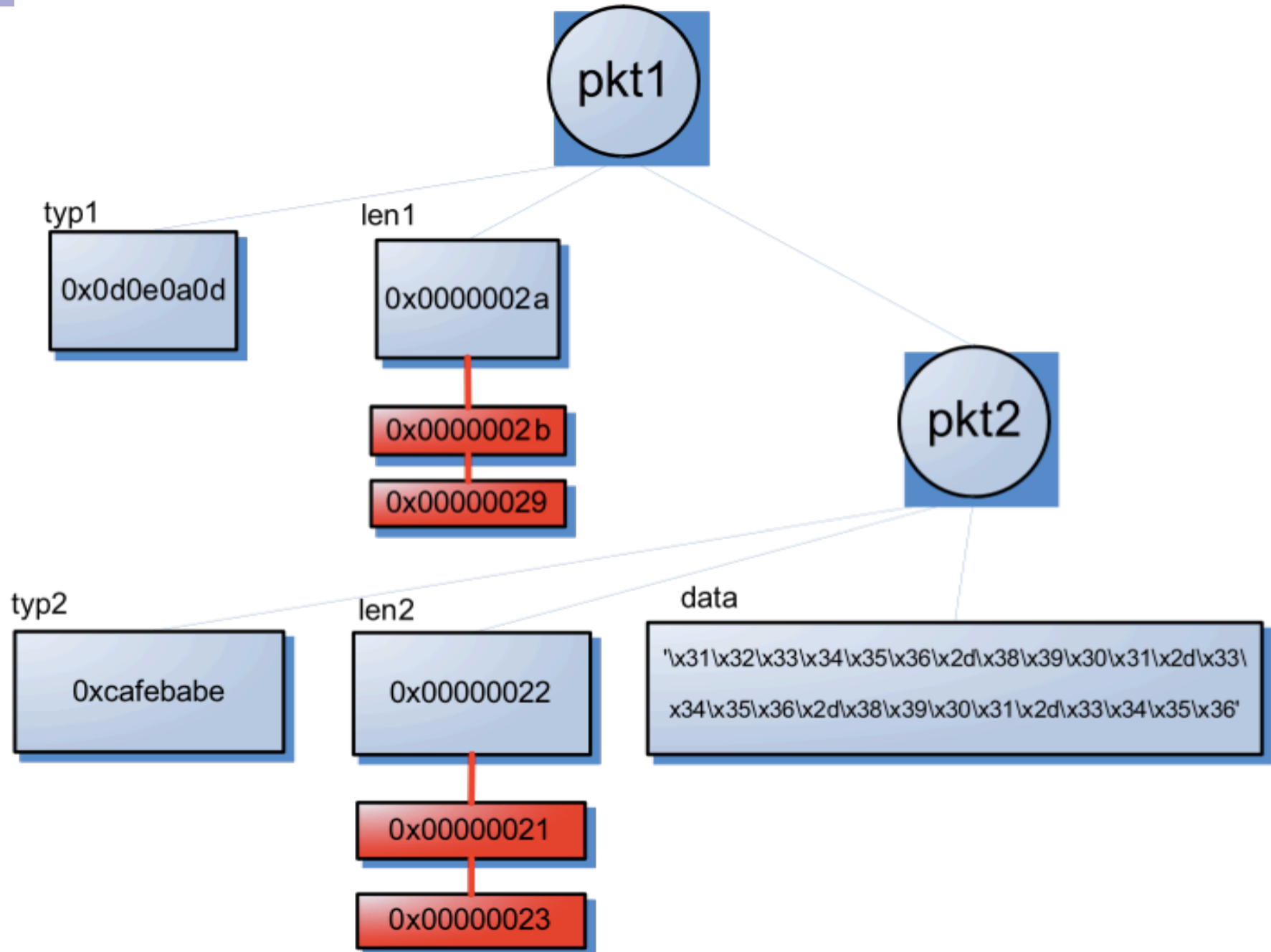
# Graphically:



# RuxxerにおけるTLVプロトコル

```
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*は、実際にはカンマ区切りをサポートしていない (ここではスペースを意味するためにカンマを使用している)







## 「pkt1」の合計9つの順列

“\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”

## 「pkt1」の合計9つの順列

“\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”

## 「pkt1」の合計9つの順列

“\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”

## 「pkt1」の合計9つの順列

“\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”

## 「pkt1」の合計9つの順列

“\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”

## 「pkt1」の合計9つの順列

“\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”

## 「pkt1」の合計9つの順列

“\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”

## 「pkt1」の合計9つの順列

“\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”



## 「pkt1」の合計9つの順列

“\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”



# 「リソースの問題」

- 多くのネストされた構造を持つ複雑なプロトコルは、無数の反復につながる可能性がある
- データを生成するfuzzerは、予め可能性のあるすべての順列を拡張しているため、「リソースの問題」を生じ、しばしばメモリの枯渇につながる
- RuXXerは、ダイナミックにオブジェクト属性を操作し、オブジェクト属性に過負荷をかける



# その他のRuXXerの機能

- 反復機能への高速転送
- さまざまなGUI機能
  - 「ファイルからバイトを挿入」
- Commの拡張性
- 言語インタプリタの拡張性

# 結論

- ダムなプロトコル非対応のfuzzingは十分ではない
- 既存のfuzzingフレームワークは、機能のために使い勝手を犠牲にしており、また使い勝手のために機能を犠牲にしている
- RuXXerは、強力なfuzzingフレームワークの上部にシンプルな言語を配置することで



# <http://www.ruxxer.org>

- RuXXerバンドル (またはソース) のダウンロード
- RuXXerアップデートの入手
- RuXXer Wikiの参照
- RuXXer SVNレポジトリの閲覧
- バグ/機能リクエスト/アイディア/ブレインストームの提出



# 質問 / コメント

- 電子メール:

- [stephen@ruxxer.org](mailto:stephen@ruxxer.org)

- [colin@ruxxer.org](mailto:colin@ruxxer.org)