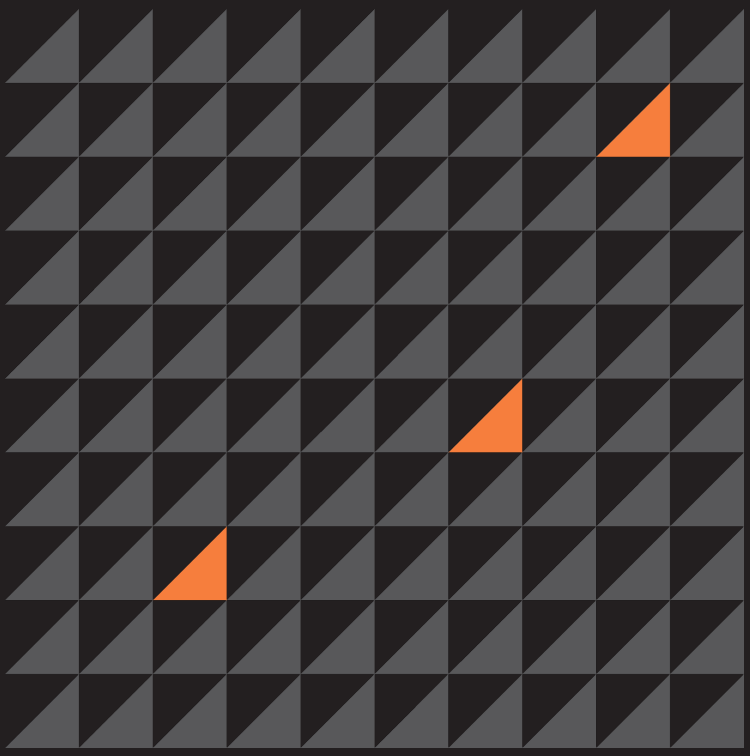
Public
EXTERNAL

Volatility Past & Present

Creating a more stable volatility

Mike Auty - Senior Security Researcher
MWR Infosecurity



Contents

Why now?

Why work on Volatility 3.0 instead of 2.4?

- Big breaks are tough on the community
- Volatility 2 has limitations
- Offers the opportunity for large changes

We can make changes to:

- Coding style
- Core Interface
- Internal architecture

What was so bad about volatility 2.x?

Amazingly, nothing major!

However, Volatility 2.x was originally written:

- With only XP support
- Before we knew how the object model would work out
- With certain assumptions for simplification
- Many years ago!

The Volatility Generic Architecture

Three primary components:*

- Objects
  - These defined how to read and interpret chunks of data from memory
- Address Spaces
  - These defined how to map from one chunk of memory to another
- Profiles
  - These held all the symbols, in a single space, and some architecture-specific details (x86, x64, arm, etc)

* Using the Volatility 2.x naming scheme

Goals for Volatility 3.0

Stable API

For volatility 3.x, we want:

- Python 3.x
- A well-designed, well-defined interface
- Fewer limitations imposed by the codebase
- Speed improvements
- Explicit error handling

Basically, improving from past experience!

Public
EXTERNAL

Objects

The cookie cutters

Objects

Volatility 2.x Proxied Objects

- Always read the data live
- Tried very hard to be resilient
- Proxied all standard calls to look like an object

It had some drawbacks:

PYTHON

```
class BaseObject(object):
  def v(self):
      ...
  def m(self, memname):
      ...
  def d(self):
      ...
```

- It made overloaded operator ordering important
- We have lots of code with **int(VolObject)**

Objects

Proposed Objects

Native Python Types

- Turns out you can inherit from Native types, but...
  - NativeTypes are immutable
  - Requires overriding __new__

An example, the proposed Volatility Integer:

PYTHON

```
class Integer(PrimitiveObject, int):
    """Primitive Object that handles standard numeric types"""
    def __new__(cls, context, layer_name, offset, symbol_name, struct_f
        value = cls._struct_value(struct_format, context, layer_name, o
        return int.__new__(cls, value)
```

Objects

Exceptions & None Objects

NoneObjects were designed as a convenience

- Any error led to a *NoneObject* where all methods returned itself

This was a really lazy "solution":

- Still had to check for NoneObjects rather than exceptions
- No one knew how to diagnose the issues, so still required debugging
- Didn't solve any of the problems it was supposed to!

From now on:

- We always throw exceptions when necessary
- The developer has to be aware of things like zread...
  - ...and their consequences!

Objects

Constructing objects

Previously objects were curried

- All objects were traversed at profile load
- Made debugging difficult
- Required constructing dummy objects to determine values

Currying is now explicit by using *ObjectTemplates*

- They can hold attributes (eg, structure size)
- They can be interrogated for their contents
- *ReferenceTemplates* allow delayed symbol lookup

Objects

Proposed API - Objects

PYTHON

```python
class ObjectInterface(validity.ValidityRoutines):
    """ A base object required to be the ancestor of every object used
    def __init__(self, context, layer_name, offset, symbol_name, size,
        """Initialize the object"""
    def write(self, value):
        """Writes the new value into the format at the offset the objec
    def cast(self, new_symbol_name):
        """Returns a new object at the offset and from the layer that t
```

PYTHON

```python
class Template(object):
    def __init__(self, symbol_name = None, **kwargs):
        """Stores the keyword arguments for later use"""
        self.arguments = kwargs
        self.symbol_name = symbol_name
    def update_arguments(self, **newargs):
        """Updates the keyword arguments"""
    def __call__(self, context, layer_name, offset, parent = None):
        """Constructs the object"""
```

Objects

Proposed API - Templates

PYTHON

```
class ObjectTemplate(interfaces.objects.Template, valid                yRout
    def __init__(self, object_class = None, symbol_name = None, **kwarg
        """Initializes the object with the object class and symbol_name
    @property
    def size(self):
        """Returns the size of the template"""
    @property
    def children(self):
        """A function that returns a list of child templates of a templ
    def replace_child(self, old_child, new_child):
        """A function for replacing one child with another"""
```

PYTHON

```
class ReferenceTemplate(interfaces.objects.Template):
    """Factory class that produces objects based on a delayed reference
    def __call__(self, context, *args, **kwargs):
        template = context.symbol_space.resolve(self._symbol_name)
        return template(context = context, *args, **kwargs)
```

Address Spaces

The dough

Address Spaces

Nomenclature

The name's pretty good, but people didn't know how to work with them

- For example, the original CrashDump was a CrashDumpFile Address Space
- People couldn't grasp what vtop (v-top?) meant very easily

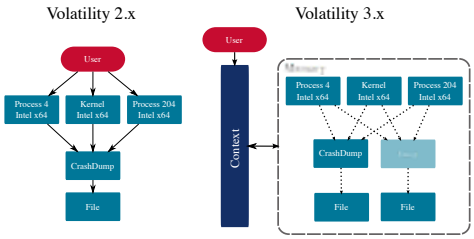
A more accurate name would be *TranslationLayers*

- A TranslationLayer can have multiple sources
- Leaf-nodes are called *DataLayers*

The layers will be grouped in a *Memory* container

Address Spaces

Multiple Sources



Address Spaces

Proposed API

PYTHON

```
class DataLayerInterface(validity.ValidityRoutines):
    def __init__(self, context, name):
        """Initializes the DataLayer with a name"
    def is_valid(self, offset):
        """Returns a boolean based on whether the offset is valid or no
    def read(self, offset, length, pad = False):
         """Reads an offset for length bytes and returns 'bytes' (not 's
    def write(self, offset, data):
         """Writes a chunk of data at offset"""
```

PYTHON

```
class TranslationLayerInterface(DataLayerInterface):
    def translate(self, offset):
        """Returns a tuple of (offset, layer) translating of input doma
    def mapping(self, offset, length):
        """Returns a sorted list of (offset, mapped_offset, length, lay
    def dependencies(self):
        """Returns a list of layer names that this layer translates ont
```

Public
EXTERNAL

Profiles

The kitchen Drawer

Profiles

Design

Profiles weren't named well. What is a Profile?

- A self-referential dictionary/table of symbols
- Information about the architecture

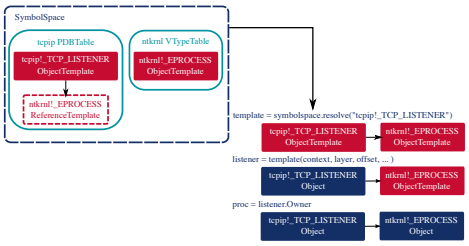
Profiles are now *SymbolSpaces* made up of individual *SymbolTables*

- Only one SymbolSpace at a time
  - ...built-up as necessary from the config
- Each SymbolTable can have its own native table
- CStructs members will stay as strictly offset-based

This should allow us to mix and match table implementations

Profiles

Symbol Tables



Profiles

Proposed API - SymbolSpace

PYTHON

```
class SymbolSpace(collections.Mapping):
    def __init__(self, native_symbols):
    """Handles an ordered collection of SymbolTables"""
    @property
    def natives(self):
        """Returns the native_types for this symbol space"""
    def resolve(self, symbol):
        """Takes a symbol name and resolves it (dealing with ReferenceT
    def append(self, value):
        """Adds a symbol_list to the end of the space"""
    def remove(self, key):
        """Removes a named symbol_list from the space"""
```

Profiles

Proposed API - SymbolTables

PYTHON

```
class SymbolTableInterface(validity.ValidityRoutines):
    def __init__(self, name, native_symbols = None):
        """Handles a table of symbols"""
    def resolve(self, symbol):
        """Resolves a symbol name into an object template"""

    @property
    def symbols(self):
        """Returns an iterator of the symbol names"""
    @property
    def natives(self):
        """Returns None or a symbol_space for handling space specific n

    ... set/get/del symbol_class ...

    ... readonly dict methods ...
```

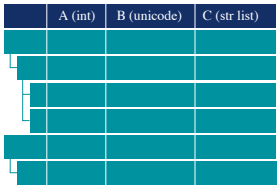
Looking Ahead

The Rest of the Kitchen

Looking Ahead

Unified Plugin Output

List/Tree Hybrid



Looking Ahead

Same Old Problems

- Instantiating objects in the wrong layer
  - Leaning towards requiring explicit dereferencing of pointers
  - Possibly attach the native layer to the Memory object?
  - Go whole hog and allow multiple "location" layer/offset pairs?
- Architecture information and metadata
  - What sort of information do plugins require to work?
  - Which is associated with the architecture rather than the OS?

Looking Ahead

More of the Same Old Problems

- Allowing developers the freedom to modify the framework
  - Their changes may be useful to lots of plugins
  - Don't want people pushing "use volatility and my patches"
  - Don't want people disrupting existing plugins
- Handling of 64-bit pointers
  - Maybe easier now that pointers **are** integers
  - Either chop the bits off the top...
  - ...or follow the sign extension rule

Summary

Summary

- Volatility 3.x is already in development!
- There will be some big changes, but hopefully all for the best
- Just the tip of the iceberg:
  - Plugin framework
  - Unified Plugin Output
  - Configuration
  - Command Line Interface
  - Scanning framework
  - Memory Factories
  - More complex objects
  - Convert all the core plugins
  - etc

Public
EXTERNAL

Questions?

email mike.auty@gmail.com
twitter @volatility
www volatility.googlecode.com/