Reverse Engineering Python Applications

Aaron Portnoy and Ali Rizvi-Santiago, TippingPoint DVLabs

aportnoy@tippingpoint.com , arizvisa@tippingpoint.com

Abstract

Modern day programmers are increasingly making the switch from traditional compiled languages such as C and C++ to interpreted languages like Ruby and Python. These types of languages are gaining popularity due to their flexibility, portability, and ease of development. This paper is a study of the Python language and methods by which one can leverage its intrinsic features to reverse engineer and arbitrarily instrument applications. It will cover techniques for interacting with a running interpreter, patching code both statically and dynamically, and manipulating type information. The concepts are further demonstrated with the use of AntiFreeze, a toolset for visually exploring Python binaries and modifying code therein.

Introduction

This paper focuses on some of the overlooked risks that come along with the use of a dynamic language. The current methods for distributing closed-source dynamic language code have not been extensively dissected publicly by reverse engineers. This instills a notion among developers that their source code is safe when distributed in the binary forms provided by the language. However, the use of a dynamic language tends to make reversing applications written in it even easier than static languages.

Dynamic languages can be defined as high-level languages that perform type checking at runtime. Many also have support for reflection, metaclasses, and runtime compilation. These features are of particular interest to the reverse engineer as they require a significant amount of type information to exist in the distributed application. In the following sections we will discuss how these aspects of dynamic languages are implemented in Python and some of the more interesting things that can be accomplished by leveraging them.

Python Code Object and Byte Code Primer

To fully understand the inner workings of Python, one should first become familiar with how Python compiles and executes code. In Python, when code is compiled the result is a code object. A code object is immutable and contains all the relevant information needed for the interpreter to run the code. In order to present an example we will first introduce the properties of a code object [1][2]:

- *co_argcount*: An integer representing the number of arguments to be passed to the object.
- *co_nlocals*: An integer representing the number of local variables and arguments used.
- *co_stacksize*: An integer denoting the required stack size to execute this object.
- *co_flags*: An integer value whose bits denote the existence of specific properties.
- *co_code*: A string representing the byte code.
- *co_consts:* A tuple containing constant values referenced by the byte code.
- *co_names*: A tuple of strings containing names used.
- *co_varnames*: A tuple of strings containing names of local variables used.
- *co_filename*: A string containing the file name from which this code object was compiled.
- *co_name*: A string that contains the name of the code object if it has one.
- *co_firstlineno*: An integer denoting the line number this object begins at in the source.

- *co_lnotab*: A string mapping offsets in the byte code to line numbers in the source.
- *co_freevars*: A tuple containing names of free variables.
- *co_cellvars*: A tuple containing names of local variables referenced by nested functions.

The *co_code* property is the most significant as it contains the byte code instructions for the code object. The byte code itself does not contain data but rather index references into other code attributes such as the *co_consts* tuple.

Python byte code consists of a set of 113 one-byte opcodes [3]. A byte code instruction is represented as a one byte opcode value followed by arguments when required. As mentioned above, data is referenced using an index into one of the other properties of the code object.

For example, given this byte code string:

```
'\x64\x02\x64\x08\x66\x02'
```

The first byte of the byte code string is 0x64 which maps to the instruction LOAD_CONST. The LOAD_CONST instruction takes a single one-byte argument. The next instruction is the same opcode, but with a different argument. The final instruction is opcode 0x66 which maps to BUILD_TUPLE. The BUILD_TUPLE instruction takes a single argument. Thus, the disassembly of this byte code is simply:

```
LOAD_CONST  0x02

LOAD_CONST  0x08

BUILD_TUPLE 0x02
```

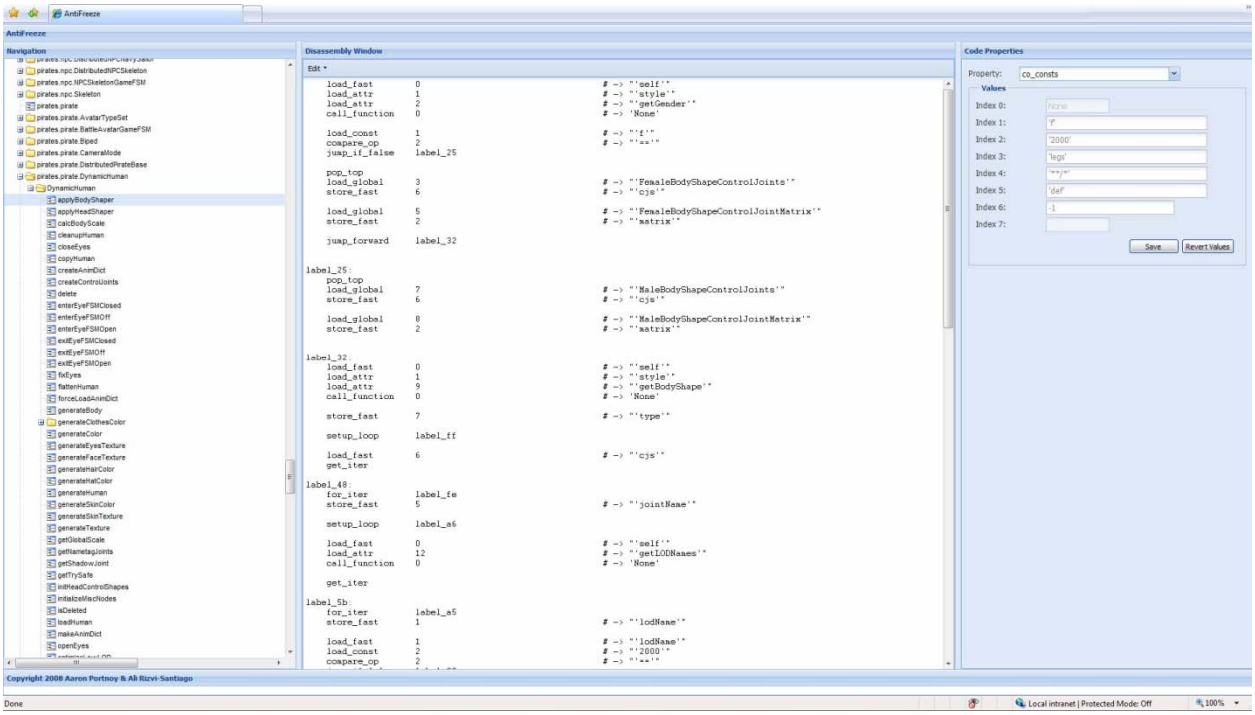
Python byte code operates on a Last-In First-Out (LIFO) stack of items. The LOAD_CONST instruction pushes a constant value on to this stack. As mentioned previously, Python byte code references data using an index. The 0x02 argument to LOAD_CONST instructs the interpreter to fetch the item at index 0x02 from the *co_consts* tuple and push it to the top of the stack. The LOAD_CONST 0x08 will likewise push the value at index 0x08. The final instruction, BUILD_TUPLE, creates a tuple from consecutive items popped from the stack, the number of which is specified by the BUILD_TUPLE argument. In this example, a tuple containing two objects is created and saved to the top of the stack.

Now that the basics of code object structure and Python byte code have been covered, the next section will deal with how these code objects are stored in binary form and methods by which they can be extracted.

Marshalling and Unmarshalling

Python code can be distributed in a multitude of binary formats. This is achieved using the marshal module [4] included with Python. This module provides functionality to serialize objects into binary form, as well as extract them. The most commonly encountered binary format is a compiled Python file (.pyc). These files contain a magic number, a timestamp, and a serialized object. This file type is usually produced by the Python interpreter in order to cache the compiled object to avoid having to parse the source multiple times.

The next most complex format that Python is commonly distributed in is known as a frozen Python module and is produced by the freeze.py tool [5] distributed with CPython. Each object within the given source is compiled to a code object and then serialized. The freeze.py tool will produce a collection of C source files containing the serialized objects defined as an array of structures. Each structure is composed of a table containing the object name, a pointer to its data, and a length value. The developer will compile this code into a shared object which then becomes de-serialized once the frozen library is initialized by the Python interpreter.



[Figure 1]: AntiFreeze GUI Screenshot

Throughout the remainder of this paper we will be focusing on the frozen format.

Code Object Modification

In Python many internal objects, including code, are immutable during runtime. However, due to Python's ability to utilize reflection, one can clone a code object by re-creating it, passing the same parameters the original had. In order to do this the code object type must first be attained. This can be accomplished with the following line of code:

```
code = type(eval("lambda:x").func_code)
```

This line of code creates a function and retrieves the type of the function's func_code property. Now that the type has been saved, we can instantiate an object by calling the code constructor. The prototype for a code object is as follows:

```
code(argcount, nlocals, stacksize,
flags, codestring, constants, names,
varnames, filename, name, firstlineno,
lnotab[, freevars[, cellvars]])
```

At this point, we can re-create the original code object, modifying its properties and their values if desired. To replace a code object within a compiled or frozen file, one would need to re-serialize the newly created code object using the marshal module and write it back into the data structure it was extracted from.

AntiFreeze

To streamline this process we have developed a toolset for interacting with frozen Python files. AntiFreeze is a web-based utility that enables one to easily browse, modify, and inject code objects from a PYD file. It provides the ability to view and edit disassembled Python code directly, as well as edit any properties of a code object. The toolset is comprised of four major components: functionality for extracting code objects, a disassembly engine, a Python assembler, and the interface itself. Figure 1 shows the interface layout which was built upon

the Ext-JS JavaScript library [6]. The pane on the left of Figure 1 is a tree view representing the hierarchy of objects. For example, if a given node represents a Python class, its child nodes could be sub classes or functions defined within its scope. Once an object is selected from the left pane, the center pane is updated with that object's disassembly. This field is editable so that a user may edit instructions or data indices by hand and re-assemble if they so wish. The far right pane allows a user to inspect and edit any of the current code object's properties. The data for these components is obtained via the disassembly engine portion of the toolset. The code is a rewrite of the dis module [7] distributed with CPython with some useful additions including the display of de-referenced data and code location labels.

Through the interface a user is also able to re-assemble their new code object at which point the value of the center editor is passed to our Python assembler, along with any changed code properties. The assembler performs a simple 2-pass scan and, if valid, the code object becomes assembled and injected to replace the old object within the PYD file.

This process allows for quick and efficient modifications to PYD files without the considerable hassle of other methods.

Python Code Object Execution

In order for Python to execute a code object, the object must be bound to its locals and globals. This is done by the Py_EvalCode function exported by the Python shared library. In addition to binding references, this function is responsible for creating an internal type known as a Frame. Frame objects are handled by the Py_EvalFrame [8] function which is responsible for processing the object’s byte code.

The Py_EvalFrame function is also responsible for maintaining the state of a global locking mechanism known as the Global Interpreter Lock, or GIL [9]. This lock is used to govern the operation of multiple threads that run under the context of a single process. When performing any runtime modifications, the state of the GIL must be taken into account.

Python Specific Features

Due to Python's support for introspection and the fact that all objects are first-class [10], the Python interpreter has to know about all currently executing objects. This requires a substantial amount of information be available about an object's methods and properties during runtime. This information can then be leveraged by a reverse engineer to achieve a variety of objectives.

Another interesting feature of the Python interpreter is the ability to re-evaluate code. This essentially means that the language's compiler has to exist in memory. This feature can best be leveraged by a reverse engineer by calling the function PyRun_SimpleString from the context of the interpreter a user would want to inject code into. The process to do so begins by obtaining the GIL. Then, one would call PyRun_SimpleString and release the lock afterwards. Being able to inject code into the Python interpreter allows one to execute any code of their choice. This enables a reverse engineer to perform a multitude of useful tasks such as logging all function calls using sys.settrace [11] and utilize ihooks [12] for hooking tasks performed internally by the interpreter.

Anti-Reversing

Python's ability to construct a code object during runtime increases its ability to withstand some of the above reversing techniques. Due to the ability to construct a code object with user-supplied arguments, one can store the *co_code* attribute in any format. During execution transformations can be performed upon the property. Then, it could be assigned it to a function object or an instance method object which can then be executed by the interpreter. To take it even further, the object itself could call back into the Python interpreter by utilizing the functionality provided by the ctypes [13] module included with Python 2.5. The first step of this

[Figure 2]: ToonJumpForce Cheat Screenshot

approach would be to obtain the type returned by the function sys._getframe. At this point one could create a frame with custom parameters. Once the frame is created, it can be executed by passing it to the Py_EvalFrame function via a ctypes function prototype.

Case Study: Pirates of the Caribbean Online MMORPG

To demonstrate the impact of the discussed techniques, this section will present a case study involving injecting cheats into a popular multiplayer online game.

Disney's Pirates of the Caribbean Online [14] is a multiplayer online RPG game written in Python and the majority of the code is distributed within a PYD file. Using AntiFreeze, we can quickly begin to browse the object hierarchy and determine which code objects we would like to edit. A quick perusal of the namespace turns up the promising 'pirates.piratesbase.PiratesGlobals' object. Once this object is selected, the disassembly window is updated with about 3000 lines of code. One particular section starting at line 897 of this disassembled code appears interesting:

```
load_const  161  # '24.0'
store_name  196  # "'ToonForwardFastSpeed'"
load_const  161  # '24.0'
store_name  197  # "'ToonJumpFastForce'"
load_const  162  # '8.0'
store_name  198  # "'ToonReverseFastSpeed'"
load_const  163  # '120.0'
store_name  199  # "'ToonRotateFastSpeed'"
load_const  161  # '24.0'
store_name  200  # "'ToonForwardSpeed'"
load_const  161  # '24.0'
store_name  201  # "'ToonJumpForce'"
```

These instructions simply load constant values from the *co_consts* tuple and store them to variables within the code. To edit these values, we can choose *co_consts* from the drop-down menu of properties and edit accordingly. In this example, we will change the *co_consts* at index 161 from 24.0 to 99.0 and see if our in-game character behaves differently. Editing the field within the interface and choosing assemble

automatically injects the updated values into the PYD file and it's ready to be tested.

The screenshot in Figure 2 confirms that the injection worked as our in game character is able to jump considerably higher than normal. This quick demonstration illustrates how AntiFreeze can quickly apply static code object modifications to frozen Python code.

Conclusion

The choice of a dynamic language can lead to many overlooked and sometimes undesirable side effects. As their popularity continues to grow, there will likely be a demand for these languages to design mechanisms to protect proprietary code. Until then, however, their features will continue to be susceptible to the design flaws outlined herein.

References

[1] Code Properties, http://effbot.org/pyref/type-code.htm

[2] Code Properties, http://www.voidspace.org.uk/python/weblog/arch_d7_2006_11_18.shtml

[3] Opcodes, http://svn.python.org/projects/python/trunk/Lib/opcode.py

[4] Marshal Module, http://svn.python.org/projects/python/trunk/Python/marshal.c

[5] Freeze, http://wiki.python.org/moin/Freeze

[6] Ext JS, http://extjs.com/

[7] Dis Module, http://svn.python.org/projects/python/trunk/Lib/dis.py

[8] Py_EvalFrame, http://svn.python.org/projects/python/trunk/Python/ceval.c#514

[9] GIL, http://wiki.python.org/moin/GlobalInterpreterLock

[10] First-class Object, http://en.wikipedia.org/wiki/First-class_object

[11] Sys.settrace, http://docs.python.org/lib/debugger-hooks.html

[12] ihooks, http://pydoc.org/2.4.1/ihooks.html

[13] CTypes, http://docs.python.org/lib/module-ctypes.html

[14] Disney's Pirates, http://disney.go.com/pirates/online/

[15] Reversing Engineering Dynamic Languages RECON 2008, http://www.recon.cx/2008/speakers.html#python