Reverse Engineering Python Applications

Aaron Portnoy
TippingPoint DVLabs
Ali-Rizvi Santiago
TippingPoint DVLabs

Abstract

Modern day programmers are increasingly making the switch from traditional compiled languages such as C and C++ to interpreted dynamic languages such as Ruby and Python. Interpreted languages are gaining popularity due to their flexibility, portability, and ease of development. However, these benefits are sometimes counterbalanced by new security exposures that developers are often unaware of. 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. We 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 new toolset we present for visually exploring Python binaries and modifying code therein.

1. Introduction

Dynamic languages are 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. This paper focuses on some of the potential security implications that arise when releasing software written in a dynamic language. Because the current methods for distributing dynamic language code in binary form have not been publicly scrutinized by reverse engineers, there might exist a notion among developers that their source code is relatively concealed when distributed in the binary forms provided by the language. However, their software is in fact more susceptible to reverse engineering than a compiled program written in a static language.

In the following sections we will discuss how these aspects of dynamic languages are implemented in Python, specifically CPython, and some of the more interesting reverse engineering techniques that can be accomplished by leveraging their functionalities.

2. 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. When code is compiled in Python the result is a code object. A code object is immutable and contains all of the information needed by the interpreter to run the code. The following are some of the more notable properties of a code object [1][2]:

- *co_code*: A string representing the byte code.

- *co_consts*: A tuple containing constant values referenced by the byte code.

- *co_name*: A string that contains the name of the code object, if it has one.

The co_code property is the most significant as it contains the byte code instructions for the code object. A byte code instruction is represented as a one byte opcode value followed by arguments when required. Data is referenced using an index into one of the other properties of the code object. For example, take this byte code string:

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

The first byte of the byte code string is the opcode, 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 also 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 stack of items. The LOAD_CONST instruction pushes a constant value on to this stack. 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 property 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 case, a tuple containing two objects is created and saved to the top of the stack.

The above example demonstrates the basic process one would take to disassemble a simple code object. A more enterprising extension would be to attempt to decompile the byte code back into readable Python source code, complete with object and function names. This could be accomplished by utilizing the other properties of a code object such as the co_name, co_names, co_varnames, co_filename, and others.

3. Marshalling and Demarshalling

Python code can be distributed in binary form by utilizing the marshal module [3]. This module provides the ability to serialize and deserialize code objects using the store and load functions. The most commonly encountered binary format is a compiled Python file (.pyc) which contains a magic number, a timestamp, and a serialized object. This file type is usually produced by the Python interpreter as a cache of the compiled object to avoid having to parse the source multiple times.

Python code can also be distributed in what is known as a frozen Python module (.pyd). This format is produced by the freeze.py [4] tool distributed with CPython. Each object within a given Python source is compiled to a code object and then serialized using the marshal module. 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 so that a program written in C is able to load the file as a module.

There are other less popular binary formats that have been developed and all are created in some fashion using the marshal module. This commonality allows a reverse engineer to utilize the deserialization capabilities of the marshal module to extract code objects and all associated metadata. Once extracted, a code object can then be modified and serialized back into the binary medium.

It should be noted that the marshal module makes no claims that it secures or obfuscates source code, however this appears to be assumed by many developers as sensitive portions of code are still commonly distributed in these binary forms.

4. 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 obtained. This can be accomplished with the following line of code:

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

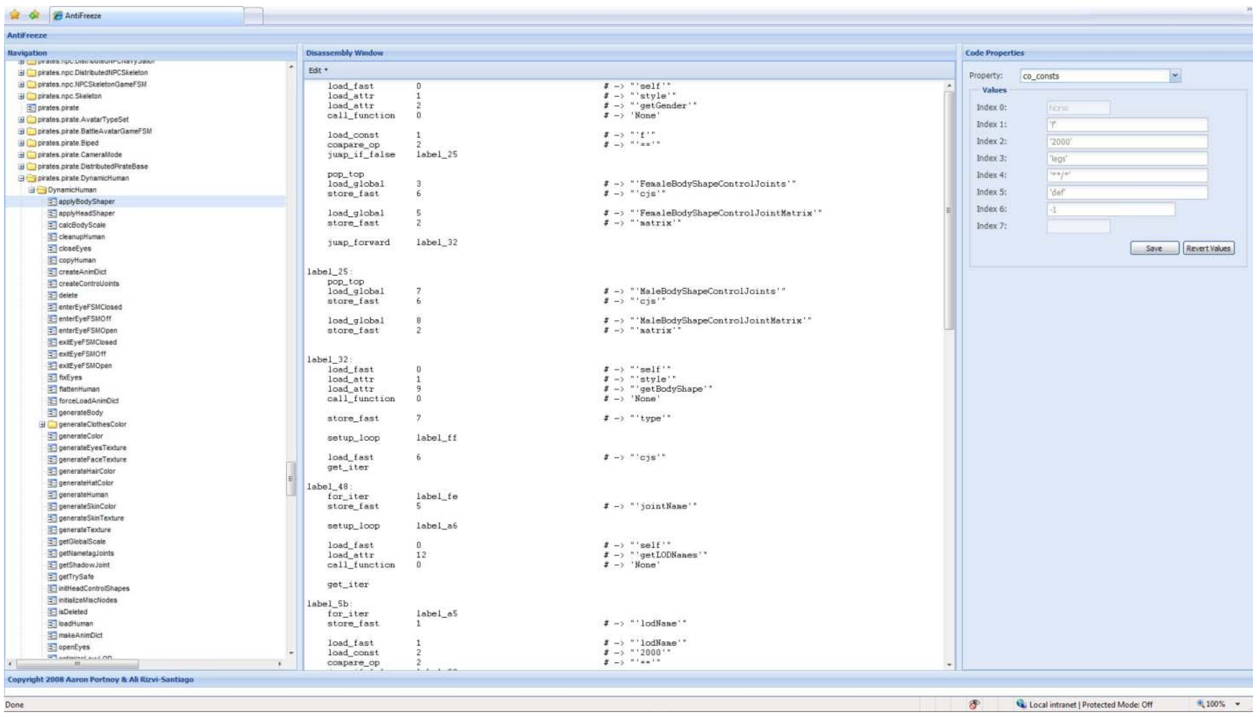
This line of code will create a function and retrieve the type of the function’s func_code property. Now that the type has been saved, an object can be instantiated by calling the code constructor. The prototype for a code object is as follows:

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

At this point, the original code object can be re-created, 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.

5. AntiFreeze

To streamline this process we have developed a toolset for interacting with frozen Python files. AntiFreeze [5]



```
pirates.npc.DistributedNPCSkeleton
pirates.npc.NPCSkeletonGameFSM
pirates.npc.Skeleton
pirates.pirate
pirates.pirate.AvatarTypeSet
pirates.pirate.BattleAvatarGameFSM
pirates.pirate.Biped
pirates.pirate.CameraMode
pirates.pirate.DistributedPirateBase
pirates.pirate.DynamicHuman
  DynamicHuman
    applyBodyShaper
    applyHeadShaper
    calcBodyScale
    cleanupHuman
    closeEyes
    copyHuman
    createAnimDict
    createControlJoints
    delete
    enterEyeFSMClosed
    enterEyeFSMOff
    enterEyeFSMOpen
    exitEyeFSMClosed
    exitEyeFSMOff
    exitEyeFSMOpen
    fixEyes
    flattenHuman
    forceLoadAnimDict
    generateBody
    generateClothesColor
    generateColor
    generateEyesTexture
    generateFaceTexture
    generateHairColor
    generateHatColor
    generateHuman
    generateSkinColor
    generateSkinTexture
    generateTexture
    getGlobalScale
    getNametagJoints
    getShadowJoint
    getTrySafe
    initHeadControlShapes
    initializeMiscNodes
    isDeleted
    loadHuman
    makeAnimDict
    openEyes
```

```
    load_fast       0                            # -> "'self'"
    load_attr       1                            # -> "'style'"
    load_attr       2                            # -> "'getGender'"
    call_function   0                            # -> 'None'

    load_const      1                            # -> "'f'"
    compare_op      2                            # -> "'=='"
    jump_if_false   label_25

    pop_top
    load_global     3                            # -> "'FemaleBodyShapeControlJoints'"
    store_fast      6                            # -> "'cjs'"

    load_global     5                            # -> "'FemaleBodyShapeControlJointMatrix'"
    store_fast      2                            # -> "'matrix'"

    jump_forward    label_32


label_25:
    pop_top
    load_global     7                            # -> "'MaleBodyShapeControlJoints'"
    store_fast      6                            # -> "'cjs'"

    load_global     8                            # -> "'MaleBodyShapeControlJointMatrix'"
    store_fast      2                            # -> "'matrix'"


label_32:
    load_fast       0                            # -> "'self'"
    load_attr       1                            # -> "'style'"
    load_attr       9                            # -> "'getBodyShape'"
    call_function   0                            # -> 'None'

    store_fast      7                            # -> "'type'"

    setup_loop      label_ff

    load_fast       6                            # -> "'cjs'"
    get_iter

label_48:
    for_iter        label_fe
    store_fast      5                            # -> "'jointName'"

    setup_loop      label_a6

    load_fast       0                            # -> "'self'"
    load_attr       12                           # -> "'getLODNames'"
    call_function   0                            # -> 'None'

    get_iter

label_5b:
    for_iter        label_a5
    store_fast      1                            # -> "'lodName'"

    load_fast       1                            # -> "'lodName'"
    load_const      2                            # -> "'2000'"
    compare_op      2                            # -> "'=='"
```

Property: co_consts

Values

Index 0: None
Index 1: 'f'
Index 2: '2000'
Index 3: 'legs'
Index 4: '**/*'
Index 5: 'def'
Index 6: -1
Index 7:

Copyright 2008 Aaron Portnoy & Ali Rizvi-Santiago

Figure 1: AntiFreeze GUI Screen Shot

is a set of utilities that enables one to easily browse, modify, and inject code objects from binary Python code. 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 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 the 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 original file.

6. Execution of a Code Object

The preceding sections demonstrate how code logic can be obtained statically. This section deals with methods by which we can gather runtime information to aid in reverse engineering an application. When Python executes a code object it must first be bound to its locals and globals 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 function which is responsible for processing the object's byte code. Both Py_EvalCode and Py_EvalFrame are convenient places to hook execution during runtime to produce code flow data. This type of dynamic information can also add a meaningful level of comprehension to the reverse engineering process.

Due to Python's support for introspection and the fact that all objects are first-class [8], the Python interpreter must know about all currently executing objects. This requires a substantial amount of information be available about an object's methods and properties

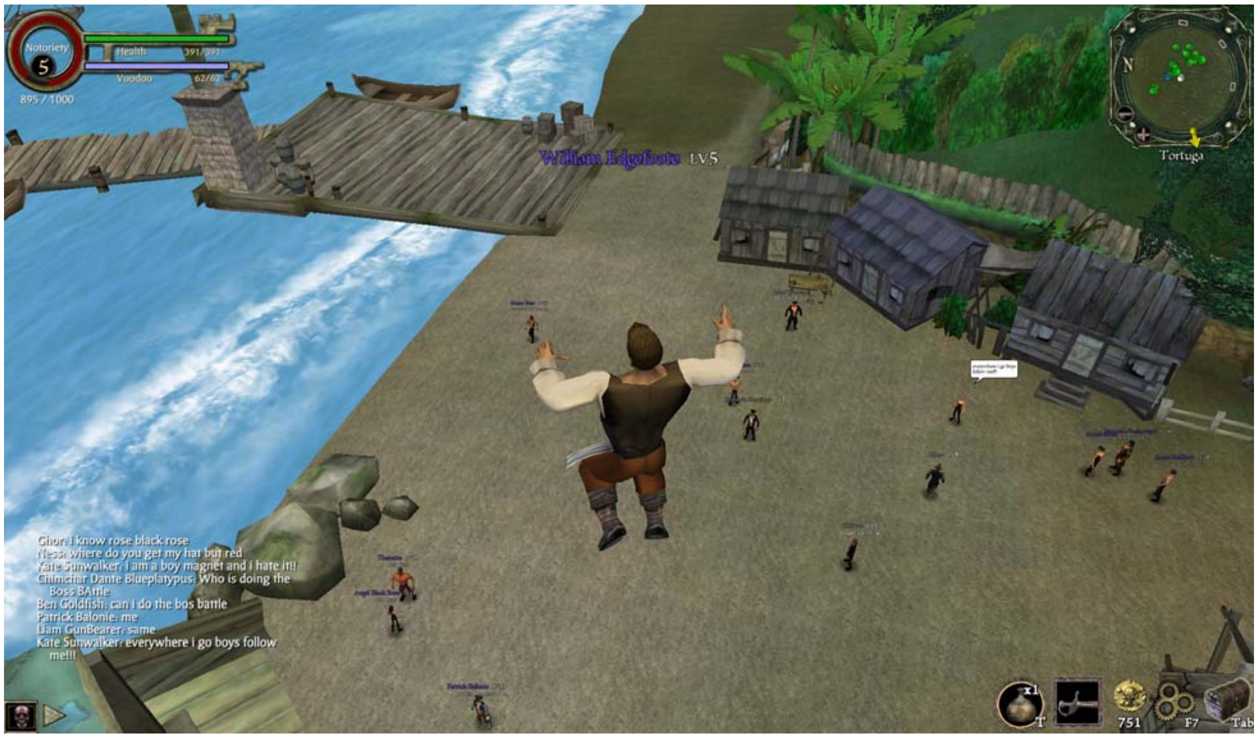

Notoriety
5
Health 391/391
Voodoo 62/62
895 / 1000
William Edgefoote LV5
Tortuga
N
751
F7
Tab
Chor: i know rose black rose
Nessa: where do you get my hat but red
Kate Sunwalker: i am a boy magnet and i hate it!!
Chimchar Dante Blueplatypus: Who is doing the Boss BAttle
Ben Goldfish: can i do the bos battle
Patrick Balonie: me
Liam GunBearer: same
Kate Sunwalker: everywhere i go boys follow me!!!

Figure 2: ToonJumpForce Cheat Screen Shot

during runtime. This information can also be leveraged by a reverse engineer to achieve a variety of objectives.

Aside from passive actions, while within the context of the Python interpreter, one can execute arbitrary code. The Python interpreter has the ability to re-evaluate code which requires the compiler to exist in memory. This feature can best be utilized by calling the function PyRun_SimpleString from the context of the interpreter. The process to do so begins by obtaining the Global Interpreter Lock [9], then one would call PyRun_SimpleString and release the lock afterwards. Injecting 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 [10] and utilize ihooks [11] for hooking tasks performed internally by the interpreter.

7. 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 with the majority of the code being 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 roughly 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 load constant values from the co_consts tuple and store them to variables within the code. To edit these values using AntiFreeze, we can select 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 test if our in-game character behaves differently. Editing the field within the AntiFreeze 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 demonstration illustrates how AntiFreeze can quickly apply static code object modifications to serialized Python code.

8. Anti-Reversing

The aforementioned techniques rely on the ease of access to byte code and type information. With a code object's byte code, code logic can be modified or even replaced entirely. Extracting type information can aid in program design comprehension and identification of function and object purposes.

The obfuscation and hardening of application byte code will always be a race between the implementers and those seeking to break it. To attempt to defend against byte code retrieval, the logical first step is towards a runtime translation solution. Properties of a code object could be stored in any signed, encrypted, or otherwise obfuscated format that is de-obfuscated or translated during runtime and used to instantiate a new object. One could even change the way variable name lookups work within the interpreter to obfuscate naming information. By adding a translation layer between the lookup of the actual names and the names within the source code, a developer could further mitigate reversing attempts.

9. Conclusion

The choice of a dynamic language can introduce some unintended new security exposures to an application. As their popularity continues to grow, there will likely be a demand for these languages to include mechanisms to better protect proprietary code. Until then, however, their features will continue to introduce susceptibility to the techniques outlined in this paper.

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

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

[5] AntiFreeze, http://code.google.com/p/antifreeze/

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

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

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

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

[10] 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 Online, http://disney.go.com/pirates/online/

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