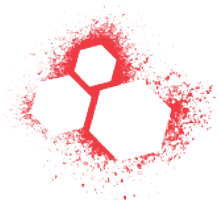


'*pyREtic*' – In memory reverse engineering for obfuscated Python bytecode

Rich Smith <rich@immunityinc.com>
Immunity Inc

Abstract

Growing numbers of commercial and closed source applications are being developed using the Python programming language. The trend with developers of such applications appears to be that there is an increasing amount of effort being invested in order to stop the sourcecode of their application being easily obtainable by the end user. This is being achieved through the use of a variety of obfuscation techniques designed to impede the common methods of Python decompilation. Another trend occurring in parallel is the use of Python as an increasingly present component of 'Cloud' technologies where traditional bytecode decompilation techniques fall down not through obfuscation, but through lack of access to the bytecode files on disk.

The techniques discussed in this paper extend existing Python decompilation technologies through taking an approach that does not require access to standard Python bytecode files (.pyc/.pyo), but rather focuses on gaining access to the bytecode through instantiated Python objects in memory and using these to reconstruct a sourcecode listing equivalent to that composed by the applications author. Approaches will also be discussed of how to defeat the common obfuscation techniques that have been observed in use in order to be able to use the in memory decompilation techniques.

Finally a proof of concept embodiment of the techniques developed will be discussed which will allow people to quickly leverage them to evaluate code for bugs that was previously opaque to them.

1. The Problem Space

The starting point for the work discussed was the need to be able to audit Python[i] applications for security relevant bugs in order to make assertions about the risk they may introduce into an environment. In the pursuit of this goal it became apparent that many closed source/non-free programs that were written in Python were making efforts to hinder the assessment of their security through attempting to not allow access to their sourcecode. Python has become an increasingly popular language of choice for a variety of commercial & closed source applications due to its ability to allow rapid development and prototyping, as well as because of the wide variety of modules available to integrate into almost any other system or protocol suite. While the underlying rationale and implications of restrictions on sourcecode availability in the context of security assessment is beyond the scope of this paper, it is fair to say that the fact that access was being restricted was enough to spur the creation of a set of technique that restored such access.

In addition, the ongoing trend in the development of Software as a Service, 'cloud' technologies and virtualisation means that for a reverse engineer many things that may have once been taken

for granted have now been turned on their head. Access to the program files of the application being reversed is no longer a given, access to applications may well be through 'walled gardens' on a remote service providers system. Such an evolution in the provision of software to the general population calls for a change in approach to it's reversing if the art of reverse engineering is to stay current.

Finally, the subject of reversing Python at the Python layer itself rather than at a lower layer (C, Java, Assembly etc) was a subject that was, by itself, an interesting one. Higher-level languages while being a boon to security in many respects are far from immune to having security flaws in the logic with which their applications are implemented, or from the developer not understanding the quirks and errata that come with every language of sufficient complexity to find mainstream adoption. The appreciation of the security implications of such high-level language artefacts can only come through working at the layer in which they exist. Immediately breaking out the debugger to look at the low level system interactions may not always be the best approach when looking for application layer / language specific bugs. The additional fact that Python is a cross platform language means that the discovery of a Python layer bugs gives a cross operating system, cross system architecture attack pathway which is often seemingly over looked.

There are a number of other well-cited areas outside of security and reverse engineering where the ability to look inside deliberately obfuscated application code holds significant value. One of the most compelling being the use of such tools to identify license violations of code that is included in closed source projects, yet has one of the many open source licenses. Detection of copyright infringement is also another obvious use case.

2. Existing Python Reversing Techniques

A discussion of any advancement in an approach should only take place in the context of what has gone before and upon which it builds. Python reversing has mainly centred around the decompilation of bytecode[ii] (.pyc and .pyo files) back to Python sourcecode (.py). It has been common practise for some time for the bytecode-compiled versions of Python applications to be distributed in order to stop the casual observer from seeing the applications internal workings.

There are a number of free and commercial options available for performing bytecode decompilation encompassing both software and services including UnPyc[iii], decompile.py[iv], decompyle[v], the related decompyle service (commercial service)[vi] and depython (commercial service)[vii]. The versions of Python upon which these above are able to work vary, as does the accuracy of the results when real complex applications are being decompiled. It is fair to say that the commercial service offerings generally have much better results, especially with the most recent Python versions, but that the free offerings are 'good enough' in many cases.

This being said, they all approach the problem in the same way. At the abstract level the approach can be summarised as taking the bytecode (.pyc/.pyo) as input, evaluating it in a static manner and producing the sourcecode representation based on that analysis.

In the context of 'lost sourcecode' this mode of operation is entirely adequate, however in the context of gaining access to the sourcecode of an application which is hosted remotely or which has been deliberately modified to evade such decompilation workflows then the requirement of access to standard Python bytecode files can be a roadblock to reversing.

The standard Python modules for disassembling (dis[viii]) and debugging (pdb[ix]) are also worth mentioning for completeness. The dis module is able to take Python objects and files and disassemble them to produce dumps of 'Python Assembly Language', but does not produce

Python sourcecode. The pdb module is the standard way in which debugging can be achieved in the Python runtime, however it is fairly basic in nature and is designed to function best when the .py file is available on disk to it. The pdb module should be squarely viewed as a tool to aid development where (by definition) sourcecode is available; its functionality quickly becomes limited when it is being run only with access to .pyc bytecode. The roles of both modules in the realisation of the new techniques discussed in this paper are included in the appropriate sections below.

3. Python Application Packaging Technologies

As Python is being used as the language of choice by an increasing number of developers, there are a number of options available with respect to the delivery of the application to the end user in the most convenient manner possible. As Python is an interpreted language requiring it's own runtime there are a number of dependencies an end user must meet before being able to take a .py or .pyc application and be able to run it. While in the GNU/Linux and BSD (and by extension OS X) worlds the inclusion of Python in the standard operating system has become almost defacto, Microsoft Windows does not support it out of the box. There is also the question of versioning and non-stdlib dependencies. All in all for a commercial application written in Python the ability to package up the application, it's dependencies and an appropriate runtime into a bundle, which the end user can 'just run', is a compelling one.

There are three main systems in use to fulfil this role targeted at each of the main OS groups. For Windows there is py2exe[x], for OS X there is py2app[xi] and for GNU/Linux & BSD's there is cx-freeze[xii]. While a detailed discussion of the techniques used by each solution is beyond the scope of this paper, it should be noted that these systems are in widespread use by commercial entities developing in Python and familiarity with them and how they store the actual application code will be of great benefit to anyone working with Python reversing. They are also the main way in which modified Python distributes runtimes, as discussed below many obfuscation techniques rely on modifying the runtime in order to achieve obfuscation.

One further note on the topic of packaging systems is that of bugs in the underlying Python runtime. As the packagers include their own Python runtime along with the application code, the chance of the packaged runtime version becoming stale over time and being vulnerable to bugs that have been discovered in the runtime is very possible. Even if the systems version of the Python runtime is patched, the bundled versions in application packages will not be. Knowing the versions of Python bundled with any packaged application is a valuable attack opportunity that is often overlooked.

4. Obfuscation Techniques

During the period in which commercial/closed source Python applications have been assessed a number of obfuscation techniques employed by authors have been noted. They range in complexity from trivial to fairly involved, the most involved being the initial impetus for the development of a toolkit to address the problems they raised. While each technique will be discussed individually they are often seen used in conjunction with one another in an attempt to raise the barrier of reversing further.

The simplest form of obfuscation and one that is very often seen is the distribution of bytecode only forms of the application. As has already been discussed the existing Python decompilers can in general easily deal with this form of obfuscation and so it will not be discussed further.

4.1 Hiding in a packager/Tying to a modified runtime

The superficial alteration of an attribute of a packaged runtime such as a version string coupled with an application level check of the attribute is a trivial method to bypass. When this has been seen it is also often coupled with a much-reduced set of stdlib modules in the bundled runtime. The goal of the author can be guessed to be stopping the application code from running in anything but their cut down runtime with the hope being that dynamic/runtime analysis will be disrupted. If runtime analysis is desired then a simple understanding of the packing system used is all that is required to get access to the Python runtime files. From that point taking any stdlib Python from the same main version (2.4.x, 2.5.x, 2.6.x etc) and putting it into the correct location will allow that module to be used to assess the application.
This also clearly does not guard against the static analysis done by the traditional Python decompilers on the applications .pyc files once they have been accessed inside the package.

4.2 Bytecode magic number switching

The file format of the Python bytecode in the .pyc/.pyo is deliberately undocumented[xiii] to allow for the changes of format between versions without the hassle of things which relied on the previous format breaking. Despite this the format is well understood to consist of the following[xiv]:

- A 4 byte magic number (with the last 2 bytes 0xD, 0xA)
- A 4 byte timestamp modification timestamp
- A marshalled code object

The magic number is used to determine which runtime version is the correct one to run the bytecode under and changes with every version number change. For CPython all the currently known values can be found in the comments at the top of import.c[xv] from the underlying runtimes sourcecode. If the magic number does not match the version of the Python runtime used to invoke the bytecode then the following error message is generated:

```
RuntimeError: Bad magic number in .pyc file
```

If the bundled runtime is modified to have a different set of version to magic byte mappings then an arbitrary non-standard value can be used as the magic byte value. This will mean that any standard Python runtime will now refuse to run the bytecode. This also affects many of the traditional decompilers as they are expecting correctly formatted bytecode and use the magic number to determine what version of Python bytecode they will analyse. If an unexpected value is provided then many will refuse to go any further in their decompilation.

A simple solution to such obfuscation is to change the magic bytes to one of the standard values. If the exact version of the bytecode is not known there is a small enough set of valid magic bytes to mean that trying them all until one works is not out of the question.

4.3 Marshal format change

Just as with the alteration to the runtime to use a different magic number, the entire marshalling format used to encode the code object in the .pyc/.pyo can be altered in a packaged runtime. This means that any other standard python runtime will not understand how to interpret the bytecode file into valid python code objects, as well as meaning that the Python decompilers will also fail. Depending on the complexity of the changes made to the marshalled code object within the .pyc or the structure of the .pyc itself will determine whether the obfuscated format can be converted back to a standard form.

4.4 Bytecode encryption

Bytecode encryption can essentially be thought of as a more complex variant of changing the marshalling format. However the key difference is that understanding the changes is much more difficult requiring access to some form of embedded secret in the modified Python runtime to gain access to the Python assembly instructions. Once again such a change to the bytecode format on disk means that a standard Python runtime is unable to run the bytecode, and traditional decompilers cannot perform its task. A small number of instances of what are assumed to be bytecode encryption as opposed to a different marshalling scheme have been seen, however none have as yet been decrypted to validate proof positive this assumption. This is an area where attacking the lower layers of C and assembler would likely have good success, but as this research was focussing on Python layer reversing this has not been investigated fully as yet.

4.5 Sourcecode obfuscation

Another type of obfuscation that will be mentioned for completeness is the obfuscation of the source itself[xvi]. This approach does not try and alter the bytecode, but creates an alternate source listing which has equivalent functionality to the original but is much more complex to follow, typically through using various types of indirection and functional segmentation. The philosophy clearly being 'have the code, you won't understand it'. While this is a popular choice with javascript (especially in websites hosting malware) the author has not seen a real world use case of this manner of obfuscation with Python, though that is obviously not to say some do not exist.

4.6 Opcode remapping

Opcode remapping is where the modified Python runtime takes the standard value to operation mapping which associate a Python assembly language mnemonic with a unique value and switches them around. This means that even if the bytecode is available, a decompiler will produce gibberish sourcecode output (if it manages anything) as the stream of bytecode is being interpreted with incorrect meanings being assigned to the Python assembly operations.
In standard Python the opcode mapping is found in the opcode.py[xvii] module, which is fairly straightforward to follow. The modified Python runtime will not ship with this file, and the reliance of other modules such as dis on opcode.py , and the subsequent reliance of decompilers on dis and a correct opcode map can cause some headaches.
The technique of opcode remapping has been seen on numerous occasions and proves to be an effective, yet easy to implement deterrent to reverse engineering. It is often seen used in conjunction with other techniques described above, and as such stops traditional decompilation techniques with no simplistic work around.

It was the presence of opcode remapping in conjunction with other techniques in an application that was begging to be assessed which initiated this research. The specific methodology used to defeat this is discussed below.

5. A new approach to Python reversing and decompilation

All of the Python obfuscations previously discussed, with the possible exception of sourcecode obfuscation, have evolved to combat the normal decompilation workflow of taking standard Python bytecode files and taking the instructions found within back to a sourcecode form. As soon as the decompiler is unable to understand the input provided to it then the game is over.

The more generic a solution is in dealing with obfuscated bytecode the better, continually playing a cat and mouse game is not a desirable goal. Identifying common approaches used in all the discussed obfuscations, as well as hopefully the unobserved ones, should conceptually allow for a new reversing approach to defeat all the existing anti-reversing techniques in use for Python.

One such common approach to all the existing anti-reversing mechanisms is that the aim is to protect the application while it is at rest, while it is sitting in files on disk. The variety of modified runtimes seen all have a set of secrets they use to 'unlock' the bytecode in order for them to be able to get things back into a standard Pythonic form which is executable. That is to say, once the program is running in memory it has already been stripped of its protection by the very mechanism that was protecting it.

Taking a running Python application that has been arbitrarily obfuscated on disk and producing the sourcecode with which it was coded was the most generic way in which all the anti-reversing techniques could be circumvented. The specific manner in which this was achieved is discussed below. A proof of concept tool was written to demonstrate the feasibility of the ideas discussed, as well as to address the actual problem that was set out to be solved – looking for bugs in closed source/commercial Python applications.
While this is the same approach as many compiled language debuggers use to access the internals of a running program, it must be remembered that the end goal is to get back to a representational sourcecode listing of the application not merely to evaluate it at runtime in a dynamic manner.

5.1 Getting 'in process'

Any methodology that relies on the evaluation of a program at runtime obviously relies on getting into the context of the running application. Even with anti-reversing techniques in use, getting into the main thread of any Python application is surprisingly simple.
When Python modules are referenced in code the .py, .pyc or .pyo file extension is not used; if there is only a single .py /.pyc/.pyo file present then that file will be used, if there are multiple files present then the timestamp embedded in the bytecode file itself (see section 4.2) is check against the mtime on the .py file to see if they are equal, if they are the already bytecode compiled .pyc/.pyo is used, if they are not it is assumed that the .py has changed and is therefore used (see the `check_compiled_module()` function in import.c [xv]). The upshot being that even if an application only shipped with obfuscated .pyc bytecode files, simply renaming one of these files and placing a .py file of the original name in it's place means that file is executed instead.
Of course the earlier in the execution sequence the file that is being replaced is, the earlier in the applications execution control is taken. Many of the application packaging technologies have manifests which state the first python file to be executed and these can often make good candidates. A further requirement is to try and allow the normal execution flow of the application to continue, albeit under control. The content of the replacement file can be anything however including something similar to the code snippet in Fig 1 allows for the replacement file to blindly mirror the functions in a file it is replacing when it is called from other areas of code.

So getting in process is easy, from here it is simple to be able to do dynamic runtime analysis, but sourcecode is the end goal being strived for.

Fig 1. Code snippet to blindly mirror a renamed module at runtime

```
import renamed_module

for x in dir(renamed_module):

    if x[:2] == "__":
        continue

    print "%s mirroring %s.%s"%(x, renamed_module.__file__, x)
    exec("%s = renamed_module.%s"%(x,x))
```

5.2 Evaluating instantiated objects

Now access to the running context of the application is available the objects in memory can be evaluated. In Python all method and function objects have a `func_code` object that contains a wealth of data about its operation and implementation. It is this object that the inspect[xviii] module uses when it is providing information about objects. The `co_code` member of the `func_code` object contains the bytecode of the function; such bytecode could be dumped and decompiled in the traditional way (albeit a function at a time).

To be able to construct the content of an entire module, the various components have to be traversed and reconstructed into a whole. There are a couple of different general method to do this a 'memory relational' approach and a 'filesystem relational' approach.

The memory relational approach takes an imported module object and inspects its attributes, each attribute is then recursively inspected until a leaf node is reached. During this traversal each method, function and generator have their code objects interrogated for bytecode with the sourcecode hierarchy being based on the memory hierarchy.

The filesystem relational approach differs in that the filesystem location of the obfuscated application is traversed looking for Python modules, once located they are imported and their attributes inspected as with the previous approach. The sourcecode produced is reconstructed in hierarchy based on that of the filesystem rather than running application. Fig 2 shows how the different object types in Python relate to each other and to a bytecode representation. In general the filesystem approach is more use, but only when access to the filesystem is available (see section 6).

Fig 2. Python types and their relations

```
method:
        im_func ->  function:
                                func_code -> code:
                                                        co_code ->   bytecode
                                gi_code     -> code:
                        generator:
```

This approach works against all of the obfuscation techniques discussed except for the opcode remapping approach. How to get access to usable bytecode that has been subject to opcode remapping is discussed in the next section.

5.3 Opcode remapping circumvention

When opcode remapping has been used the content of the `co_code` object is obfuscated in the same way as the bytecode on disk. Runtime access to the object has been achieved but this does not actually gain very much leverage as the operations to which the bytecode pertains are still opaque. To get to the point of being able to decompile such bytecode back to source a slightly more complex approach is required.

As of Python 2.6.4 there are 119 defined opcode values from which all Python applications are constructed. In order to be able to successfully decompile the remapped bytecode, the value of each remapped Python opcodes needs to be deduced. As runtime access to the modified Python has been achieved it can be used to help achieve the task of decoding via use as an oracle.

An opcode remapped runtime ships with the set of stdlib Python relied on by the application, and possibly others. All of these files when compiled from the .py to .pyc also then have the obfuscated bytecode in addition to any other obfuscation applied. However the advantage that is available here is that they came from an already known source that is freely available.

This means that if two sets of bytecode can be produced for the same sourcecode, one standard and one obfuscated, they can be diffed to yield the value to which each opcode has been remapped. Of course it is unlikely that every Python opcode will be contained within a single module so the exercise needs to be repeated across multiple modules until all opcodes have been seen or there are no more stdlib modules left.

If opcode remapping has been used by itself the marshalled code objects in the .pyc files on disk can be diffed for this purpose, if however other obfuscation has been used such a remarshalling or encryption then the streams of bytecode need to be generated from a runtime context.

It is fairly straightforward to produce two equivalent streams of bytecode for diff analysis, the following simplified process must be done twice – once in a stdlib runtime and once in the modified runtime:

1. import a stdlib module
2. get access to an ordered list of its functions/methods/generators through the `dir()` function
3. dump the bytecode from the `co_code` of each function
4. concatenate the function bytecode in the order of the `dir()` list

Even though such streams are not identical to the unmarshalled code object in a .pyc this does not matter, all that matters are that the streams represent the bytecode in standard and obfuscated form for the same functions. For the sake of clarity such ordered concatenations of a modules functions bytecode will be termed .pyb files.

Once both sets of .pyb files have been generated it is simple to compare them one byte at a time. As it is only the values of the opcodes themselves that have been remapped if the bytes compared are the same it can be assumed that the byte represents an argument value to an opocde. If the compared bytes differ it can be assumed that the values represent the remapping of one opcode value to another. Fig 3 illustrates this with a simple example. The obvious caveat is that if opcode remapping has been performed across only a subset of the opcodes some will have the same values in both the standard and obfuscated opcode sets. In practise this is fairly

easy to detect and compensate for when the new opcode map is being created looking at the bytes which follow the bytes being evaluated.
Once a new opcode value map has been created a new opcode.py can be created with the new values. The obfuscated bytecode in the `co_code` objects is now able to be disassembled/decompiled at will.

Fig 3. Simple worked example of bytecode streams with remapped opcodes being diffed

Take for example the following Python expression:

```
print “bugs”
```

In standard Python this compiles to the following series of Python Assembly instructions:

```
0 LOAD_CONST               0 ('bugs')
3 PRINT_ITEM
4 PRINT_NEWLINE
5 LOAD_CONST               1 (None)
8 RETURN_VALUE
```

These instructions in turn are represented by the following byte stream:

```
0x64, 0x0, 0x0, 0x47, 0x48, 0x64, 0x1, 0x0, 0x53
```

The bytecode produced by an opcode remapped modified runtime using the same sourcecode input would produce a different byte stream, for example:

```
0x28, 0x0, 0x0, 0x19, 0x2e, 0x28, 0x1, 0x0, 0x12
```

Now if both of these byte streams are compared byte by byte, then it is easy to see that the opcodes identify themselves as the bytes that are different and the arguments to the opcodes are the bytes that are the same:

```
LOAD_CONST      0x64 -> 0x28
[ARG]           0x0
[ARG]           0x0
PRINT_ITEM      0x47 -> 0x19
PRINT_NEWLINE   0x48 -> 0x2e
LOAD_CONST      0x64 -> 0x28
[ARG]           0x1
[ARG]           0x0
RETURN_VALUE    0x53 -> 0x12
```

From this is it easy to see the values to which the 4 different opcodes have been remapped. Continuing this process for other byte streams that are known to have been produced from the same underlying source means that the opcode map can be built up to a point where a new opcode.py can be produced for use by disassemblers and decompilers.

6. Reversing in '*The Cloud*'

Software that is delivered as a service is a trend that has been increasing over the last few years, the term 'cloud' has also been increasingly (mis)used to describe many of the systems and services providing such software. While a general discussion about this trend and the associated hype and spin is beyond the scope of this paper, general points about the impact of this on reverse engineer may not be.
As the separation between user and software continues to increase, it is not inconceivable to imagine a time when a user will not have access to the application they are using files. At this point of the traditional approaches to reverse engineering fall down and the understanding of the inner workings of an application take on the form of a blackbox web assessment. However, being able to take an object from running memory and get to a sourcecode representation of it helps to shift things back into the domain of the reverse engineer. This is where the memory relational reconstruction approach becomes useful, as there is no filesystem structure to relate things to.

Granted the usefulness of this depends on the role of the application and how the application is exposed in 'a cloud', as well as the amount of access a user has to interact with it. The higher-level takeaway from this though is that even if an application's files are not available to a user, it may be possible in high interaction services that allow for programmatic interaction through a language such as Python that the source will be obtainable. This is an area that is interesting and will hopefully be explored more fully in future. For now consider this an interesting side benefit of techniques developed to solve a different set of problems.

7. pyREtic – a proof of concept toolkit

/paɪˈrɛt ɪk/ - [pahy-ret-ik]
defn*: –adjective*
of, pertaining to, affected by, or producing fever.

The principles & techniques that have been described in this paper have been embodied into a proof of concept toolset named pyREtic that will be released at Black Hat. The decompilation part of the toolset relies on a modified version of the freely available UnPyc[iii] decompiler that is able to take unmarshalled bytecode from memory or dumped .pyb files and produce .py sourcecode. It also includes a number of bug fixes to the UnPyc project that will be contributed back into the project. However the techniques discussed should be just as applicable to any Python decompiler which can be modified to expect .pyb style bytecode rather than the marshalled .pyc format.

Tools to be able to determine the values of remapped opcodes in modified runtimes are also included. There are also various extensions to the standard pdb module that make it more use for dynamic analysis and reverse engineering when the sourcecode files are not available. This will enable people to analyse Python applications that were previously opaque to them in order to make assessments about their security.

The toolkit will be made available from the Immunity Inc website[xix].

8. Future Directions

The work discussed will be extended in future to address any new anti-reversing techniques that may develop. The toolkit will be developed to both improve the accuracy of decompilation that is

provided by UnPyc, as well as the intelligence with which the constituent parts of an in memory module are reconstructed into a whole in the form of sourcecode. The toolkit will be freely available, with members of the community being encouraged to modify it as required to meet their needs.

Up until now only CPython in the 2.x branch has been examined with respect to reverse engineering, as this is the currently most popular choice with application developers and where the need for such abilities lay. There is no reason however that as and when the need arises the concepts discussed should not be extended to the CPython 3.x branch, or indeed another Python implementation entirely such as Jython[xx] or IronPython[xxi] etc.

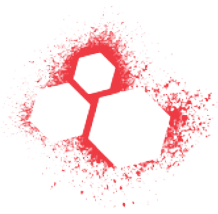
A general area of interesting research is also how to evaluate software and reverse it back to source when its files are not locally accessible, this is as true for other languages as well as Python. Future work will be conducted into the various systems where limited access to Python or a subset of it is provided to work with on a remote computing resource. The possibilities regarding the reversing and assessment of such environments will be looked at in light of the work discussed and the possibilities it raises for the acquisition of sourcecode from an instantiated object.

9. Conclusion

A generic set of techniques has been discussed, and a proof of concept embodiment of them implemented to bypass the anti-reversing techniques for Python applications that were commonly found at the time of writing. The problem of decompilation of bytecode back to sourcecode was moved from the traditional static approach where files on disk were analysed, to a dynamic approach where the application in its running state was interrogated. This created a situation where the application itself had already removed the protections it had put in place, or through access to its running context provided a means to defeat those that were remaining. Through the use of the proof of concept implementation a user is now able to go from an in-memory object to Python sourcecode representation of that object in a relatively easy manner.

Not only have the mechanisms in common use for protecting downloadable closed source and commercial Python applications from being reverse engineered been reduced, but an important first step taken into reversing Python ‘software as a service’ applications delivered even without access to their files.

Code that was once opaque to its users is now open to inspection, evaluation and risk analysis – so stop reading, **go forth and find the bugs!**



Revision History:
1.0 - 30 June 2010 – Initial version for BlackHat Vegas 2010 & Defcon 18

References:

[i] http://www.python.org
[ii] http://docs.python.org/library/dis.html#bytecodes
[iii] http://unpyc.sourceforge.net & http://code.google.com/p/unpyc/
[iv] http://users.cs.cf.ac.uk/J.P.Giddy/python/decompiler/decompiler.html
[v] http://sourceforge.net/projects/decompyle/
[vi] http://www.crazy-compilers.com/decompyle/
[vii] http://depython.net/
[viii] http://docs.python.org/library/dis.html
[ix] http://docs.python.org/library/pdb.html
[x] http://www.py2exe.org/
[xi] http://svn.pythonmac.org/py2app/py2app/trunk/doc/index.html
[xii] http://cx-freeze.sourceforge.net/
[xiii] http://docs.python.org/library/marshal.html (Paragraph 1)
[xiv] http://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html
[xv] http://svn.python.org/view/python/trunk/Python/import.c?view=markup
[xvi] http://bitboost.com/#Python_obfuscator &
http://pawsense.com/python..obfuscator/ (online demo)
[xvii] http://code.python.org/hg/trunk/file/a9ad497d1e29/Lib/opcode.py
[xviii] http://docs.python.org/library/inspect.html
[xix] http://www.immunityinc.com/resources-freesoftware.shtml
[xx] http://www.jython.org/
[xxi] http://ironpython.net/