When Pythons Bite

Alecsandru Pătraşcu[✉] and Ştefan Popa

Intel Corporation, Bucharest, Romania
alecsandru.patrascu@gmail.com, popa.stefan@gmail.com

Abstract. Python is a common used programming language in many environments, such as datacenter software, embedded programming or regular desktop computers, due to its dynamic and interpreted nature. Furthermore it is easy to write applications and test them because no recompilation is needed. At the heart of everything lies the Python interpreter which is responsible with converting input scripts into an platform-independent representation, called bytecode, and then executing them in a contained environment.

In this paper an in depth security analysis of the CPython interpreter is made. Also, a proof of concept general attack targeting the bytecode generation engine is presented and detailed. To emphasize the importance of the findings it also takes into consideration a study case on the OpenStack framework, that is widely used today in various Cloud deployments and as a software basis for many datacenters. It is chosen because it is implemented entirely in Python, rather easy to understand its internals and how to deploy it in real environments. The point made is that using our technique, or something similar, a malicious user can affect the good function of the framework, which translates into possible access gain over all the users data and applications that are stored in a Cloud environment.

Keywords: Python interpreter · CPython · Bytecode dissassembly · Bytecode infection

1 Introduction

Python is one of the most used programming languages out there today. It is a general purpose and uses a high-level programming approach. It is designed in such a manner to emphasize source code readability and to permit programmers to express their ideas using fewer lines of code than it would be otherwise necessary in languages such as C or C++. Another advantage that Python brings to table is that it is easier to debug any problems that can appear in the development and usage phase.

Because it can use various programming paradigms, such as imperative, object oriented or functional, it gained a lot of traction over time and it is now used in many projects, both small and big, such as the Django framework [1], and even as a basis for Cloud Computing deployments, under the form of OpenStack framework [2].

Being a scripting language at its core, an interpreter is needed to transform the source files (script files) into instructions that then get executed on a real processor. The canonical implementation is the CPython interpreter [3], which is a free and open-source software that benefits from a community based model of development [4].

Another thing that makes Python such a popular language is the way the interpreter manages internally the scripts. Internally, all the scripts are converted into an platform-independent intermediate representation, called bytecode. This is specific to a major version of CPython and currently we have just two implementations - bytecode for the CPython 2 or 3 family of interpreters.

In this paper is presented a top level organization of the CPython interpreter, the way it manages Python source code and how it manages to compile and execute the scripts. The security involving script execution is then approached and it is detailed the way the interpreter manages existing pre-compiled scripts and the points that makes our findings possible to use in real cases.

The structure is as follows. In Sect. 2 is presented an overview of other research that also tried to pursue this thread, emphasizing their approach and what we did different to improve it. In Sect. 3 is detailed the way CPython is working and what it internally does to execute the input scripts. In Sect. 4 it is listed the internal Python bytecode structure and how it is executed by the interpreter. Section 5 presents the proposed model used for infection of both single and multiple Python bytecode files. In Sect. 6 it is presented a study case on OpenStack and a proof-of-concept attack that target such a deployment in real case scenarios and in Sect. 7 we conclude the paper.

2 Related Work

The idea of infecting Python script files was previously studied in several white-papers or Internet blogs such as [5], but from our knowledge up to this point, there is no public mention of these approaches in the security bulletins. As we will see later, the steps needed to do it have a fair complexity to successfully gain access to a remote deployment, it works in absolute stealth mode, therefore it is very likely that such findings be in use today as 0-day infections.

One of the first documented weak points of the interpreter was in [6]. In it, the authors presents a bug existing in the Python interpreter that can theoretically permit exploiting the virtual processor in favor of an malicious user. Recent versions of the CPython try to fix it, but not completely, and in our approach we still managed to use this vulnerability, even in the latest version of the interpreter - 2.7.11.

The main problems that rise from this security perspective is given by the fact that Python can use, for speed purposes, a compiled script that exists on the computer disk, without having any mechanism to prove its origin or validity. In [7] we can see that the author tries to trigger an alarm regarding this issue and mention the fact that the interpreter runs the bytecode without additional check upon its origin or correctness. This idea represented the point of start for this paper.

An interesting piece of work is detailed in [8], which presents in detail how the interpreter starts and loads all the standard libraries. This information was used and improved in our work at the point of the initial remote infection of a remote system.

A proof-of-concept vulnerability exploit is presented in [9] and [10]. Nevertheless, their approach is limited and they do not have a real case scenario to support their findings. In this work we start the implementation following a similar path, we analyze the most interesting parts, add a more complex work flow on top of it and present an improved version, together with the possibility to infect all the bytecode files found inside a local and remote machine.

3 The CPython Interpreter

The main Python interpreter, called CPython, is the default and most widely used implementation of this programming language. It is written entirely in C and it contains an internal compiler to transform input scripts into bytecode and an interpreter to execute it at run-time. A top view representation can be seen in Fig. 1.

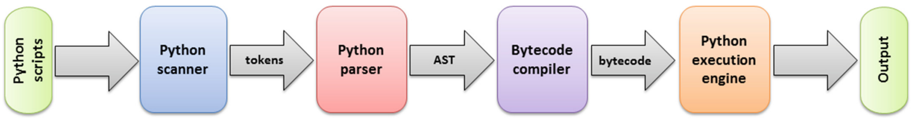
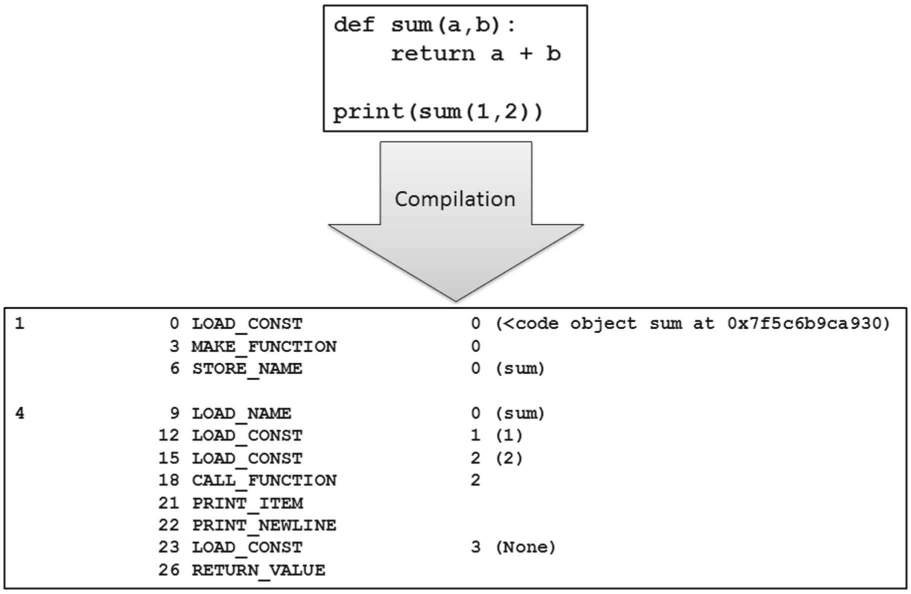


Fig. 1. CPython interpreter top level architecture

CPython features four main components, as follows. The first one, the *Python scanner* is responsible with reading the input scripts as a string stream and converting it into tokens, that are passed to the *Python parser*. The parser will create a tree internal representation of the input data, under the form of an Abstract Syntax Tree (AST). The AST is then fed to the *Bytecode compiler*, which in term converts the tree structure and its components into a stream of bytes, in an formalized structure [11]. The bytecode is then executed by the *Python execution engine*, which is in essence a virtual machine execution engine that features an internal garbage collection mechanism and various memory management modules.

An example of a simple Python input script and the corespondent bytecode can be seen in Fig. 2 and was obtained using the “dis” Python module [12].

In order to execute faster, the CPython interpreter has a feature that permits it to store the bytecode representation in a file on the computer disk. If the Python scanner detects that an input script has been already compiled into bytecode, it will directly load it and send it to the Python execution engine, bypassing the parser and the compiler. A graphical representation is depicted in Fig. 3.



```
def sum(a,b):
    return a + b

print(sum(1,2))
```

```
1             0 LOAD_CONST              0 (<code object sum at 0x7f5c6b9ca930)
              3 MAKE_FUNCTION           0
              6 STORE_NAME              0 (sum)

4             9 LOAD_NAME               0 (sum)
             12 LOAD_CONST              1 (1)
             15 LOAD_CONST              2 (2)
             18 CALL_FUNCTION           2
             21 PRINT_ITEM
             22 PRINT_NEWLINE
             23 LOAD_CONST              3 (None)
             26 RETURN_VALUE
```

Fig. 2. Python input script and bytecode corespondent

4 Bytecode Structure and the Execution Model

But why does CPython use a bytecode? The answer to this question is rather complex, but to make it simpler, the bytecode is portable, even though machine code is much faster. Creating and interpreting bytecode is a common used technique used by many other interpreters, such as Java [13] or PHP [14] among many others. Having this separate representation makes it easier to write complex interpreters based on it, other than the canonical ones. Furthermore, another advantage of it on modern CPU architectures is that the bytecode is stored in linear fashion in computer memory, thus being cache friendly.

A question may rise at this point - what other interpreters are doing to keep their bytecode safe? As mentioned above, Java uses bytecode packed in a .jar file, which is in essence a ZIP archive. The way they are implementing security at this level is to have every jar file signed with a trusted certificate and therefore every time the interpreter needs to access the bytecode will have to check the signature. On the other hand, the PHP interpreter has a feature called OPcache [15] which stores precompiled script bytecode in the memory and nothing on the computer disk.

The CPython bytecode execution engine can be described as a simple stack machine, meaning that the abstract bytecode instructions (opcodes) are using a stack for pushing and popping instructions, expressions, values and states. It features dedicated opcodes to access variables, that are used under different

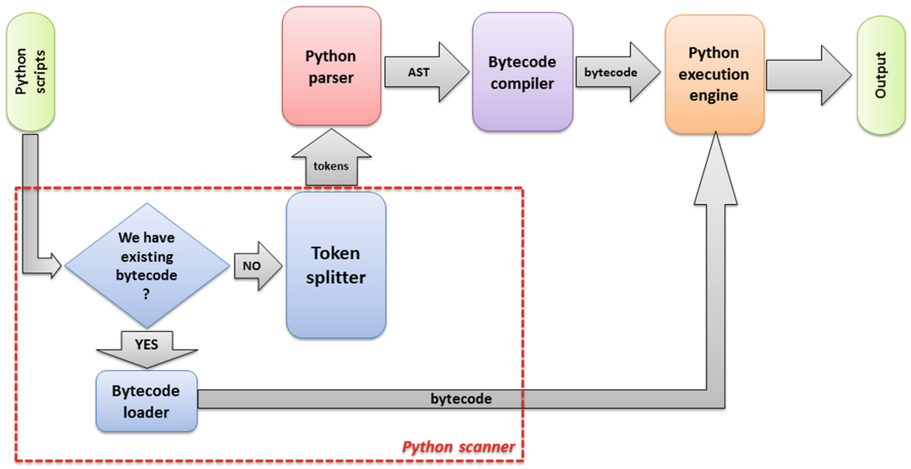


Fig. 3. Python scanner check for existing bytecode

circumstances and they look in different places. They can be split in four different families of opcodes:

– *_FAST opcodes are used to access a function local variable and are used inside a function scope. Example: LOAD_FAST, STORE_FAST
– *_DEREF opcodes are used to access variables that are used in closures. Example: LOAD_DEREF, STORE_DEREF, DELETE_DEREF
– *_GLOBAL opcodes are used for variables that are known to be global for the running script. Example: LOAD_GLOBAL, STORE_GLOBAL
– *_NAME opcodes are used for variables that are stored in Python modules or classes. Example: LOAD_NAME, STORE_NAME, DELETE_NAME, IMPORT_NAME.

The bytecode also has dedicated instructions for other things, like iterators, list creation or various standard types such as lists, numbers or strings. The generated bytecode is stored to disk, for convenience, in various formats. We can use .pyc, .pyd, .pyo, .pyw or .pyz files. But all these encapsulation have on thing in common: they do not verify the internal bytecode or its origin. They feature just a simple mechanism to detect if a script (.py file) has change, in order to re-compile the scripts and create new bytecode.

At a simple level, a .pyc file is a binary file that contains four different things:

– A magic number. This has 4 bytes in length; the first two bytes store a binary representation of the CPython interpreter needed to unserialize the stored bytecode and the last two bytes are fixed and contain the values 0x0D and 0x0A.
– A modification timestamp. This field represents the Unix modification timestamp of the source script that generated the .pyc file. This is used

to determine if the stored bytecode must be re-compiled, by comparing the script's file timestamp with the stored value.
- The pyc script size
- The serialized bytecode that CPython interpreter generated.

5 Implementation of a Pyc Backdoor

Regarding security, the Python bytecode is not secure by itself. Even if it does not allow the execution of random machine operations, it can be used to generate hand-based instruction sequence that can crash the interpreter or lead to arbitrary code execution. CPython is not implemented to be a general purpose interpreter, but it is designed to execute bytecode generated by the interpreter itself, which is guaranteed to run according to the language specification and not do other unexpected things behind the scene.

In this section we present a proof-of-concept (POC) attack under the form of Python bytecode based backdoor. Furthermore, the design of it makes it persistent and resistant to recompilation of the original script files. We present in detail the internal mechanism and the infection method.

A quick recap, we want to exploit the fact that the CPython interpreter uses a bytecode representation of every .py file its executing, in a file having the same name, but with a .pyc extension; when you run a script file, the equivalent pyc file is search in the same directory and if the timestamps match, it is executed directly.

Our POC presents itself as a self infecting payload and once a bytecode file is infected, it will automatically search for all pyc files in the current directory and infect them also. If the user modifies the source script file, the pyc will be re-generated and soon will get re-infected, as other malicious bytecode files than remain unmodified will make sure of this. Furthermore, to respect the pyc specification, every time we infect a bytecode file, we make sure that the timestamp that signs the file remains intact and we copy the original pyc size over to the new pyc file.

In order to make the code self-reproduce, a different approach must be taken. Every time a pyc is executed, it will create a list of other pyc files that exist in the same directory, and read their internal structure and scans for a dedicated marker. If the file was infected before, it is skipped, otherwise the malicious payload is copied. For the malicious payload you can use anything that can compile to a Python bytecode.

As previously stated, in order for our payload to work, it must be stored in Python bytecode format and guarded with a dedicated infection marker. For this we have chosen the magic number 0xCAFEBABE, the same magic number used by the Java interpreter, and it will be stored in a variable called marker. The generated bytecode in this case will be as follows:

```
LOAD_CONST 0xCAFEBABE
STORE_NAME marker
```

The pseudo-code for the infection mechanism can be see in the *infect_pycs* listing.

```
infect_pycs(payload)
{
    f = list_pyc_files
    for every file in f
        open file and read content
        locate the start marker 0xCAFEBABE
        if marker is found
            skip file
        else
            save timestamp
            append payload to file
            update timestamp
            save file to disk
        end
    end
}
```

A question might pop at this point - what about the pyc size? For sure, if the payload contains a lot of malicious code, infecting every .pyc file in a system will consume a significant amount of space. For modern computers this can be easily forgotten, but what about the embedded devices that have rather limited storage capabilities? To solve this issue we can reduce the payload size by using the Python capabilities to compress data offered by the zip module. An additional decompressing instruction is needed to inflate at runtime the desired payload and restore its bytecode form. By doing this we get up to 70 % reduction of payload size.

With all this information so far, a graphical representation of the modified pyc file can be seen in Fig. 4.

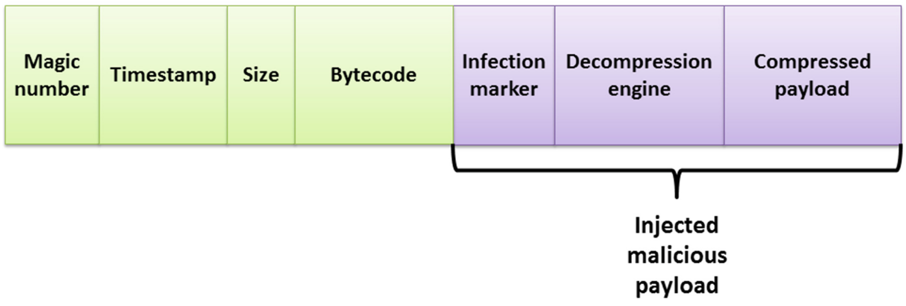


Fig. 4. Infected Pyc file

Another point worth mentioning is the fact that the user can update all the Python script files or delete all the pyc files and thus triggering an entire re-compilation. This can be done if the users suspect that the bytecode on the disk has different internal structure than the one that is generated from the script. However this risk is minimal, as most of the users do not even get notified by the antivirus software for this malicious action.

6 Study Case on OpenStack

6.1 OpenStack Overview

In this section we apply the mechanism presented in the previous section to a real case deployment scenario using OpenStack [2].

OpenStack is a widely used Cloud Computing framework that is written entirely in Python. It is a set of software tools for building and managing platforms for public and private Clouds which lets users easily deploy virtual machines and any other virtual instances that handle different tasks.

From a top level perspective, it features four main components, that can be seen in Fig. 5.

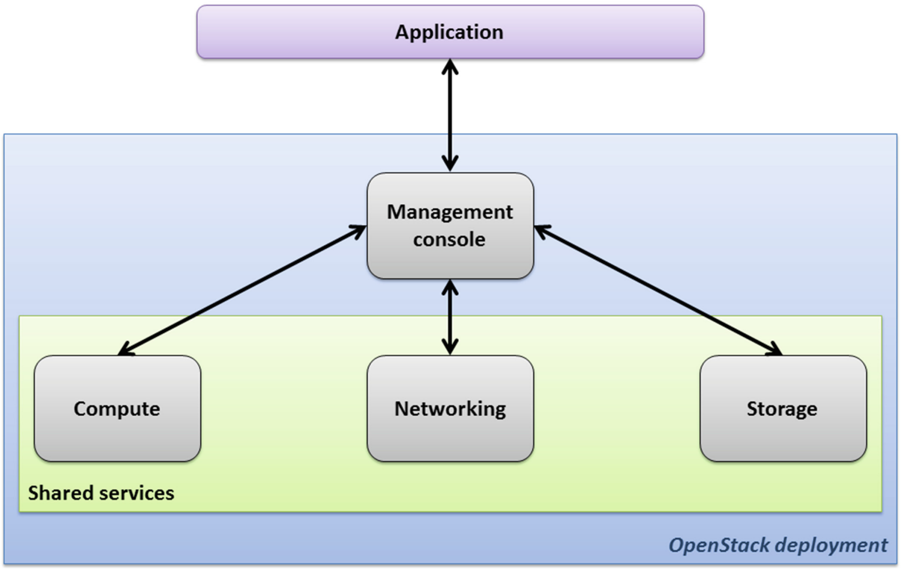


Fig. 5. OpenStack top level architecture

A software application that runs on top of OpenStack has access to its *Management console*, installed on a **master node**, from which it can start virtual environments under the form of virtual machines and/or containers. The console has a connection to the rest of the OpenStack modules and even if it does not do anything important besides acting as a front-end for the framework, it is important to mention it at this point because it acts as a centralized management unit and every administrative action, such as setting network parameters or updating the software in the Cloud deployment, is done through it.

The rest of the modules are split in three directions, based on their role in the software ecosystem. The *Compute* module is responsible with virtual instances management and monitoring them, the *Storage* module is responsible with storing virtual instances templates and other things that the running applications need, like input files. Finally, the *Networking* module assures that communication is always kept alive between all the running modules from the deployment.

OpenStack is maintained and deployed by many companies, such as Mirantis and RedHat. In this paper the focus is on one of the biggest contributor and developer of this framework - Mirantis [16]. Their approach make the entire framework easy to deploy and upgrade, with very little user intervention. This is good from administrator and user point of view, but lacks the possibility to fine tune the security details. For example, one particularity is that they deployment uses dedicated hardware which have only "root" access to their installed operating system. This means that every software application running inside it will have unrestricted access to any parts of the deployment, with full administrative privileges. This is also true for the Python interpreter, that is in discussion here.

6.2 Initial Infection

The goal in such a deployment is to infect the master node and through it to infect the rest of the deployment. In order to achieve this, an attacker can create payload with two purposes - one that keeps a connection alive with an external Command and Control server and another that scans the entire infrastructure and propagates to all the Python libraries. We discuss each of them individually in the next paragraphs.

Before detailing the first functionality, it is necessary to know several details behind an OpenStack deployment. First of all, the network connections are split based on the roles of each service. Therefore a typical deployment has a minimum of three separate network connections; the configured address is not important, as the payload can scan all the available interfaces, make a list of all of them and then attempt connection to each of the involved servers.

The second part to keep in mind is that the services are running into full administrative mode, under the user "root". Even more, for many providers, it is the only user configured to run on the host operating system, a Linux distribution in our case.

Another important detail is that the master node must have direct access to all machines that host the services. This is done by using the "ssh" application, configured with certificates for authentication. In this mode, the connection to a remote machine for the administrator or master node is simple as giving the command "ssh root@remote_ip".

Regarding the location on disk of all targeted libraries, it is not necessary to scan the entire disk to find them. Being based on Python, the target libraries are stored in a fixed location - */usr/lib/python2.7*. Walking the entire tree structure in order to find .pyc files is trivial in this case, as the Python environment offers out-of-the-box all the needed methods to do it.

After a malicious pyc is loaded into the master node, it will start by checking if already infected the files on it. It is enough to check if a magic number is found in the structure of a bytecode file. If the system was not infected before, we can apply an algorithm, as listed in the function *initial_infection*. The parameter *command_payload* represents the bytecode needed to connect to an external server, and *infect_payload* represents the code needed to recursively infect a single host, following a guideline presented in the *infect_pycs* listing.

```
initial_infection(command_payload, infect_payload)
{
    connect to an external C&C server
    report to the C&C server that acces is established
    f = list_pyc_files('/usr/lib/python2.7')
    infected_before = false
    for every pyc in f
        open pyc and read content
        locate the marker for this module
        if marker is found
            skip pyc
        else
            infected_before = true
            inject into pyc the infect_payload
        end
    end
    if infected_before
        report to the C&C server that infection was previously done
    else
        scan network interfaces
        for every interface
            save the network address and mask
            scan each of the hosts
            for every alive host
                connect through ssh
                push the infect_payload into a single remote file
            end
        end
        report to the C&C server that infection is done
    end
}
```

The attack scenario can be seen in Fig. 6. The C&C server is located outside the OpenStack deployment and the initial infected pyc, together with the two modules are located at the master. Following the red lines, we can see how the

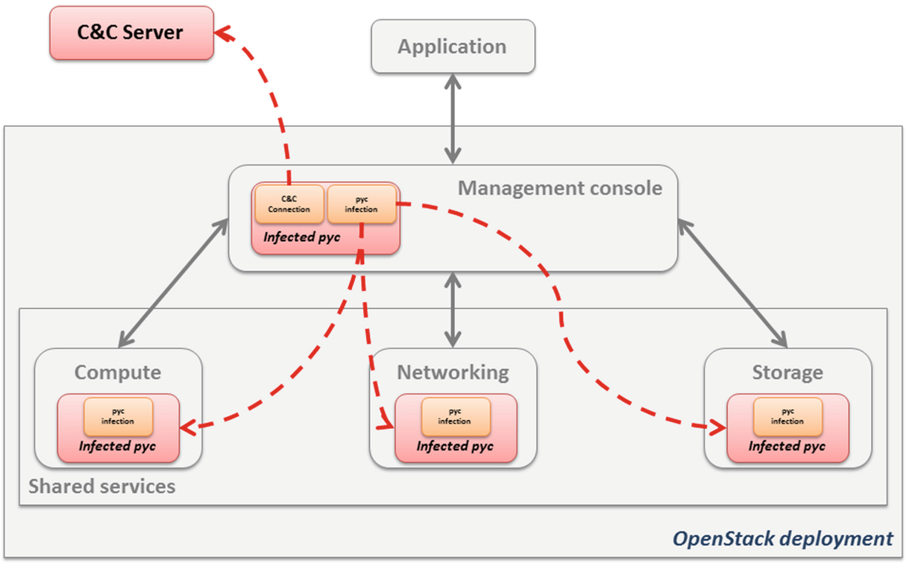


Fig. 6. Infected OpenStack deployment

malicious payload is sent to all the servers in the infrastructure. Once the infected file is ran, it will trigger the infection of all pyc files located on that host.

7 Conclusions and Future Work

In this paper a new approach was presented, that can be used for gaining unrestricted access to a workstation that is running the Python interpreter by modifying the bytecode used by it. The way the interpreter loads and uses external pre-compiled scripts represents the point of origin for the findings; also a simple code that detail the way the malicious script can be structured was presented.

To apply the findings on a larger scale implementation, a real case scenario is presented by applying our approach over an OpenStack deployment, that has the particularity of being widely used in production and being entirely written in the Python programming language. The top level architecture of this framework, together with some details regarding the way it is used in production was detailed and the previous simple code was extended to a full representation that can be used in malicious attacks.

As future work we intend to further investigate other security issues that exist in the Python interpreter and that can lead to other exploits. Of course, the notification of the Python developer community is a high priority, as this matter of insecure script loading and execution is vital to the well function of many applications and frameworks based on Python.

References

1. Django Framework. https://www.djangoproject.com/
2. OpenStack Cloud Framework. http://www.openstack.org
3. Python Main Webpage. http://www.python.org
4. CPython Interpreter Source Code Repository. http://hg.python.org/cpython
5. Python Bytecode trust. https://utcc.utoronto.ca/cks/space/blog/python/BytecodeIsTrusted
6. Python Interpreter VM. https://doar-e.github.io/blog/2014/04/17/deep-dive-into-pythons-vm-story-of-load_const-bug/
7. https://utcc.utoronto.ca/cks/space/blog/python/WhyCPythonBytecode
8. Backdooring Python via PYC. http://secureallthethings.blogspot.ro/2015/11/backdooring-python-via-pyc-pi-wa-si_9.html
9. Reversing Python Object. https://www.virusbulletin.com/virusbulletin/2011/07/reversing-python-objects#id3072912
10. Python trojan proof of concept. https://github.com/jgeralnik/Pytroj
11. Python Bytecode Specification. https://www.python.org/dev/peps/pep-0339/
12. Python Bytecode disassembler module. https://docs.python.org/2/library/dis.html
13. Java Interpreter. https://www.java.com/en/
14. PHP Interpreter. http://php.net/
15. PHP OPcache. http://php.net/manual/en/book.opcache.php
16. Mirantis OpenStack. https://www.mirantis.com/