# Beyond the Loadable Modules

Extending a Zabbix Agent for fun and profit

# Agenda

- What is the Zabbix Loadable Module ?
- The Problem. Why the loadable module is not for everyone.
- The Solution. Next step beyond the Loadable Module.
- The Requirements.
- How it is implemented ?
- What you can do with it ?
- Performance and convenience.

# What is the Zabbix loadable module.

Zabbix Loadable Module, is a simple and elegant way to extend your agent and a server.

Unlike running an external script, this is a shared library, which is loaded by Zabbix component during startup and become a part of such component, operating inside it’s address space.

So, this is not only about running a custom code from an Agent or the Server, but doing it with performance in mind.

# The Problem.

At some point, you are realized, that the performance of the *UserParameter* isn't great.

Although, Zabbix Loadable Module and correct “C” code is very powerful, it is not for everyone. Because…

**let's face the truth ….**

# The Problem.

- Most of us nowadays are not “C” developers. Which is one of the pre-requirement for developing a Zabbix Loadable Module.
- Even if we do know how and can develop using “C”, oftentimes we do not have a time to develop and “own” a piece of the “C” code.
- Even if we do know “C”, and can afford an expense to “owning” a “C” code, oftentimes we do need to develop something fast, using already existing high-level modules

# The Solution.

- Developing you Metrics collection code using language other than “C”.
- Embed this language into the Zabbix Agent by using Loadable Modules.

# The Requirements.

The solution is to use any RAD(1) language. Try to pick the one, which do have the following:

- Rich data types. OOP capabilities is a plus
- Embeddable with simple API
- Rich library of the modules, which shall allow you fast implementation of custom metrics acquisition.
- It is a accepted in your company by it's lead Architect

(1) RAD stands for “Rapid Application Development”

# The Requirements.

- Language shall be mature, well developed and documented and you shall have more than a single developer around who can code in it.
- For the people not exposed to this particular language before, it shall be fairly easy to learn
- It shall be fast enough for your purposes.
- It shall be approved by your local InfoSec team.
- It shall be readily available for all hardware and OS platforms you do need to cover.

# The Decision.

I chose the Python(1) as my “language of choice”, because it is matching of all my requirements and I do have a rich library of classes and functions which helps me to collect metrics data with ease.

(1) *http://www.python.org*

# How it is implemented.

Design decisions and a problems to solve:

- How to do the data types conversion
- How to startup and initialize
- How to route Zabbix “call for the metric” to the Python module.
- How to finalize

# How it is implemented.

Setup of the development environment:

- Ether install python-devel (or similar) package provided by your OS vendor
- Or configure and install Python from the sources. Look at *http://www.python.org*
- If you can compile any Python extension module, you are fine.

# How it is implemented.

## Python API you do need to know for the data conversion:

- PyString_FromString(...)
  Creating Python string object from “C” char*
- PyLongAsLong(), PyFloat_AsDouble(), PyArg_Parse()
  Converting Python objects into a “C” types
- PyTuple_New(...)
  Creating a new tuple. That’s how you pass parameters.
- PyTuple_SET_ITEM(...)
  Set value in the tuple.

# How it is implemented.

## Zabbix API you do need to know for the data conversion:

- SET_UI64_RESULT(…)/SET_DBL_RESULT(…)/SET_STR_RESULT(...)
  Set Unsigned Integer/Double/String as a result

- SET_MSG_RESULT(...)
  Passing a message from “C” module back to Zabbix

- get_rparam(…)
  Returning parameter from AGENT_REQUEST*

# How it is implemented.

Python API you do need to know for Initialization and Finalization:

- Py_SetProgramName(...)
  Set program name for an embedded Python
- Py_Initialize()
  Initialize Python environment
- Py_IsInitialized()
  Check if Python environment initialized
- Py_Finalize()
  Uninitialize Python environment

# How it is implemented.

## Zabbix API you do need to know for Initialization and Finalization:

- int  zbx_module_init()
  Loadable module initialization function. Executed when module loads. Returns ZBX_MODULE_OK if module loaded succedsfully, or ZBX_MODULE_FAIL otherwise. During Startup, we are calling Python function "main(...)" from module ZBX_startup.py

- int zbx_module_uninit()
  Loadable Module finalization function. Executed when Zabbix Agent goes into shutdown. Returns ZBX_MODULE_OK if module loaded succedsfully, or ZBX_MODULE_FAIL otherwise. During shutdown, we are calling Python function "main(...)" from the module ZBX_finish.py

# How it is implemented.

Python API you do need to know for proper handling of the PyObject*:

Python uses reference-based garbage collector.

- Py_INCREF(...)
  Increase reference count for a Python object
- Py_DECREF(...)
  Decrease reference count for a Python object
- Py_REFCNT(...)
  Return the number of the reference counts

# How it is implemented.

Python API you do need to know for working with Modules and executing Python “code objects”:

- PyImport_ImportModule(…)
  Importing a Python module. Returns a reference to a module.
- PyObject_GetAttrString(…)
  Lookup an attribute or an object in the Python module. Returns a reference to the attribute if found.
- PyEval_CallObject(…)
  Execute a Python “code object”.

# How it is implemented.

Zabbix API call returning list of the supported metrics:

- ZBX_METRIC *zbx_module_item_list()
  Returns a list of the metric keys

# How it is implemented.

Metrics exported from Python module to Zabbix :

- python.ping[]
  Returns “1” if module is properly initialized, otherwise “0”
- python.version[]
  Returns text representatikon of the version of the Python interpreter.
- py[…]
  Pass the call from Zabbix to Python

# How it is implemented.

## How to define metrics in Zabbix Loadable Module :

You shall define NULL-terminated array of ZBX_METRIC structures, as
static ZBX_METRIC keys[]

Each metric structure describes a single metric as:

- Name
  Name of the metric
- Flag
  Pass “0” if no parameters required or CF_HAVEPARAMS if parameters are expected
- Function
  Reference to a metric function
- Test parameters
  Test parameters to a metric function

# How it is implemented.

Example:

```
static ZBX_METRIC keys[] =
{
   {"python.ping",   0,  zbx_module_python_ping, NULL},
   {"python.version",   0,  zbx_module_python_version, NULL},
   {"py",  CF_HAVEPARAMS,  zbx_module_python_call_wrap, ""},
   {NULL}
};
```

# How it is implemented.

## Passing call from Zabbix to Python in “C”:

```
int zbx_python_call_module(char* modname, AGENT_REQUEST *request, AGENT_RESULT *result, i
    int i, retvalue_ret_code;
    PyObject* mod, *param, *fun, *ret;

    mod = PyImport_ImportModule(modname);

    if (mod == NULL) {
       SET_MSG_RESULT(result, strdup("Error importing of Python module"));
       PyErr_Print();
       return 0;
    }
    if (pass_cmd == 0)   {
       param = PyTuple_New(request->nparam);
       for (i=1;i<request->nparam;i++) {
         PyTuple_SET_ITEM(param,i,PyString_FromString(get_rparam(request, i)));
       }
    } else {
       param = PyTuple_New(request->nparam+1);
       PyTuple_SET_ITEM(param, 0, PyString_FromString(get_rparam(request, 0)));
       for (i=0;i<request->nparam;i++) {
         PyTuple_SET_ITEM(param,i+1,PyString_FromString(get_rparam(request, i)));
       }
    }

    fun = PyObject_GetAttrString(mod, "main");
    if (fun == NULL) {
       SET_MSG_RESULT(result, strdup("Python module does not have a main() function"));
       return 0;
    }

    ret = PyEval_CallObject(fun, param);

    if (ret == NULL) {
       SET_MSG_RESULT(result, strdup("Python code threw a traceback"));
       PyErr_Print();
       return 0;
    }

    Py_DECREF(mod);
    Py_DECREF(fun);

    if (PyTuple_Check(ret) && PyTuple_Size(ret) == 3) {
       long wrapper_ret_code;
       wrapper_ret_code = PyLong_AsLong(PyTuple_GetItem(ret, 0));
       if ( wrapper_ret_code == 0 ) {
         retvalue_ret_code = 0;
         SET_MSG_RESULT(result, strdup(PyString_AsString(PyTuple_GetItem(ret, 2))));
       } else {
         retvalue_ret_code = zbx_set_return_value(result, PyTuple_GetItem(ret, 1));
       }
    } else {
       retvalue_ret_code = zbx_set_return_value(result, ret);
    }
    Py_DECREF(ret);
    return retvalue_ret_code;
}
```

# How it is implemented.

## Routing call from Zabbix to Python through ZBX_call/main:

```
def main(cmd, *args):
    p_cmd = cmd.split(".")
    name = p_cmd[0]
    try:
        method = p_cmd[1]
    except:
        method = "main"
    try:
        params = tuple(p_cmd[2:])
    except:
        params = ()
    try:
        fp, pathname, description = imp.find_module(name)
        if posixpath.dirname(pathname).split("/")[-1] != "pymodules":
            ## If discovered module isn't in pymodules, we don't want to call it
            raise ImportError, name
    except:
        return (0,"Python module not exists",traceback.format_exc())
    try:
        mod = imp.load_module(name, fp, pathname, description)
    except:
        return (0,"Python module can not be loaded",traceback.format_exc())
    finally:
        fp.close()
    try:
        ret = apply(getattr(mod, method), args+params)
    except:
        return (0,"Python module threw traceback",traceback.format_exc())
    return (1, ret, None)
```

# What you can do with it.

Create metric collection. It could be something as simple as this:

ZBX_time.py

```
import time
def main(*args):
    return time.time()
```

And after you place this module in pymodules directory, you can query this metric like this:

```
[root@pnode1 ~]# zabbix_get -s 127.0.0.1 -k py[ZBX_time]
1441056212.510955
[root@pnode1 ~]#
```

# How it is implemented.

Or full-blown Python application with classes, objects and everything ...

```
import re
try:
    import psi
    import psi.process
except:
    raise ImportError, "You do not have required psi module or the one doesn't work properly"

class AFps:
    def __init__(self, cmd_patt=".+", pname=None):
        self.patt = re.compile(cmd_patt)
        self.pname = pname
        self.psl = self.getProc()
    def getProc(self):
        l = []
        for p in psi.process.ProcessTable().values():
            try:
                cmd = p.command.strip()
            except:
                continue
            if not cmd:
                continue
            if self.pname and p.name != self.pname:
                continue
            if self.patt.match(cmd):
                l.append(p)
        return l
    def n(self):
        return len(self.psl)

def main(cmd, cmd_patt, *args):
    if len(args) > 0:
        pname = args[0]
    else:
        pname = None
    afps = AFps(cmd_patt, pname)
    return afps.n()
```

# How it is implemented.

Previous example will allow you to query the metric, which returns the number of specific processes which command lines are matching a user-defined pattern. For example, that how we can query the number of sshd processes attached to pts pseudo-terminals.

```
zabbix_get -s localhost –k py["AF_ps","sshd: (.*)pts/(.*)","sshd"]
```

# Performance and Convenience

## Let's talk about performance:

- I my test, I am requesting a series of metrics from a passive Agent, using zabbix_get
- Let's test just the effectiveness of the call. Our metric will be very simple and return UNIX timestamp
- I am setting up the same module, which will be called from Loadable Module and UserParameters

In the same /usr/local/etc/zabbix_agentd.conf

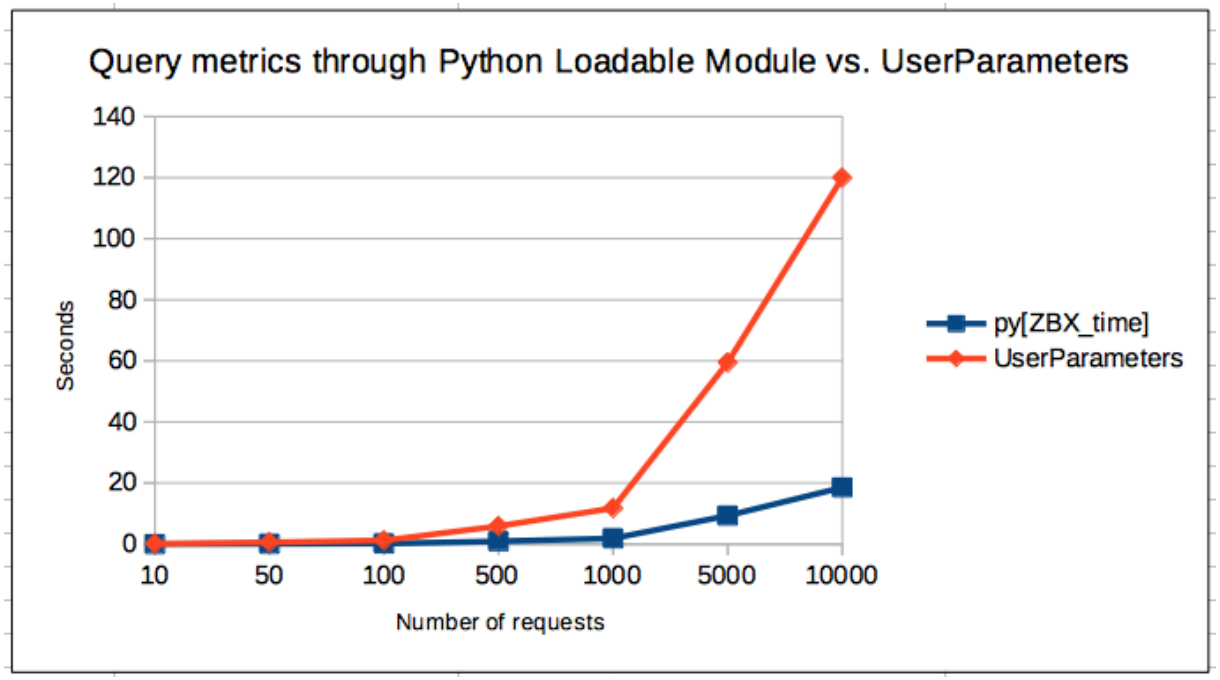
```
UserParameter=python.time,/usr/bin/python /usr/local/etc/pymodules/ZBX_time.py
```

vs

```
LoadModulePath=/usr/local/etc
LoadModule=python.so
```

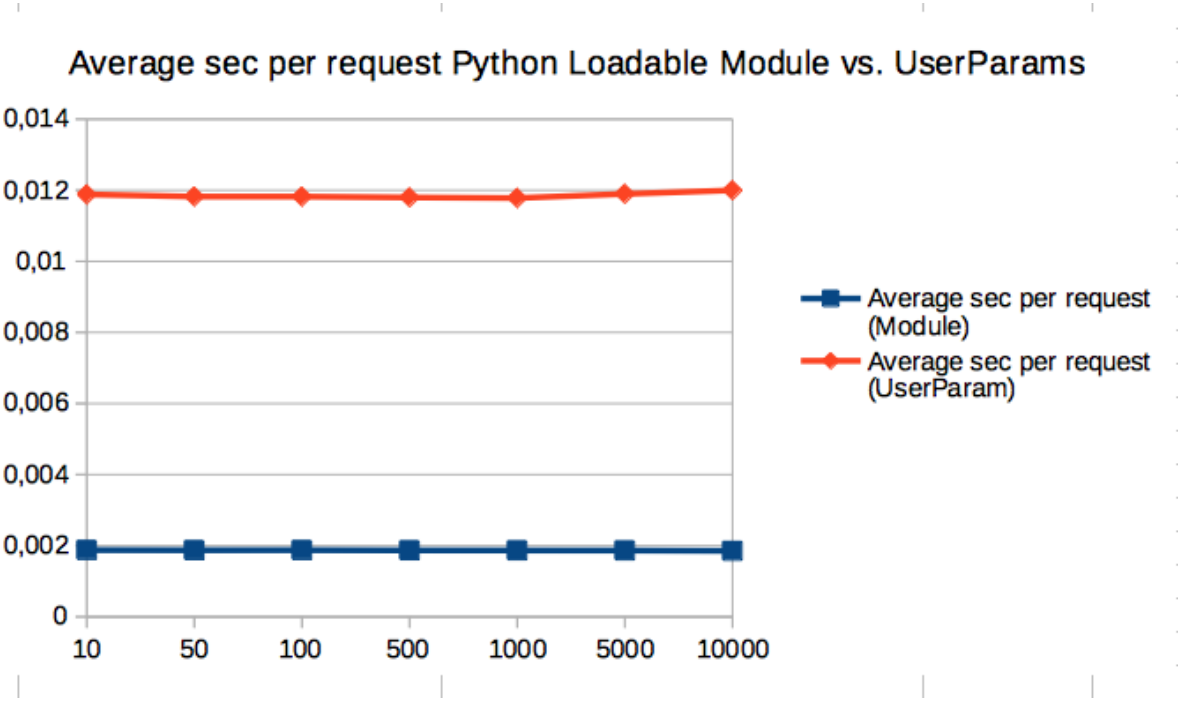
# Required time:

| | py[ZBX_time] | UserParameters |
|---|---|---|
| 10 | 0,0187578201 | 0,1188220978 |
| 50 | 0,0935981274 | 0,5911118984 |
| 100 | 0,1873850822 | 1,1819219589 |
| 500 | 0,9344751835 | 5,8981211186 |
| 1000 | 1,8665349484 | 11,7799730301 |
| 5000 | 9,3193337917 | 59,4780550003 |
| 10000 | 18,542910099 | 119,977289915 |



# Seconds per request:

| | Average sec per request (Module) | Average sec per request (UserParam) |
|---|---|---|
| 10 | 0,001875782 | 0,0118822098 |
| 50 | 0,0018719625 | 0,011822238 |
| 100 | 0,0018738508 | 0,0118192196 |
| 500 | 0,0018689504 | 0,0117962422 |
| 1000 | 0,0018665349 | 0,011779973 |
| 5000 | 0,0018638668 | 0,011895611 |
| 10000 | 0,001854291 | 0,011997729 |



# Performance and Convenience

- You are using RAD language to create a custom Metric collection
- You can use any Python features. No restrictions. But your metric collection code shall be efficient and quick.
- The test shows increase in performance. Embedded Python which loaded through Zabbix Loadable Module provides about 5.7 times faster response time while executing the same code. But your results may vary ...
- Your metric collection is as fast, as your host can compute and efficient your code.

# Conclusion


- Author: Vladimir Ulogov
  e-mail: vladimir.ulogov@zabbix.com


- https://github.com/vulogov/zlm-python

# Conclusion

Q/A ?