BLACKPHENIX

v1.0.0

OPEN-SOURCE MALWARE ANALYSIS + AUTOMATION FRAMEWORK

BPH SCRIPT DEVELOPMENT GUIDE

Developed by Chris Navarrete @ Fortiguard Labs

DOCUMENT VERSION CONTROL

Version Number	Date Issued	Update Information
V1.0	9/17/2019	First published version

Contents

Getting Started 4

What is a BPH Script? 4

What is a BPH Analysis Module? 4

BPH Script Development 5

Tool Directory Structure & Location 5

BPH Plug-In (JSON Configuration File) 5

BPH Plug-In (Special Variables) 6

BPH Plug-In (Parameters) 7

BPH Module (Python Class) 8

BPH Script (Programmatic Execution) 9

BPH Analysis Module Development 10

PEiD Walkthrough – GUI-based Application 11

PEiD – BPH Plug-in 11

PEiD – BPH Module 12

PEiD – BPH Script 13

PEiD – All together 13

Contact 13

Getting Started

BLACKPHENIX is an open-source malware analysis automation framework composed of services, scripts, plug-ins, and tools and is based on a Command-and-Control (C&C) architecture.

This framework was released and presented at BlackHat Arsenal 2019.

BPH Scripts control execution and reporting of well-known analysis tools, custom programs, and scripts that run on a virtual machine, whilst "BPH Analysis Modules" focus on data parsing, extraction, and analysis.

A malware analyst makes heavy use of tools in order to perform research activities. Many of those tools are GUI–based (e.g. ExeInfoPe, PEStudio), Console applications (e.g. pd32, DumpPE) or even scripts. However, due to the number of malware samples to analyze on a single day, it makes harder manual analysis, therefore some sort of automation will be required to overcome such a problem.

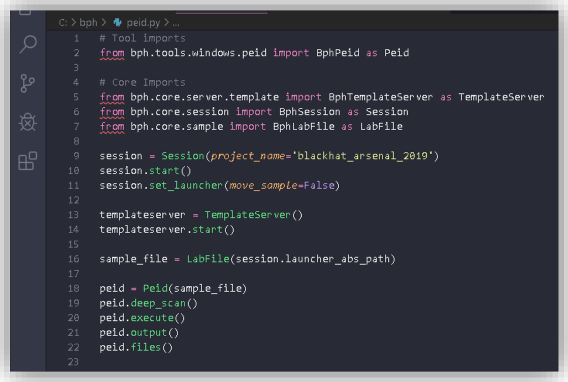
This document will provide the required guidance to introduce new tools into the framework and how to develop BPH scripts that control its execution and BPH Analysis Modules to extract and take advantage of the data generated by such tools.

What is a BPH Script?

BPH Scripts are python scripts that import python modules (**BPH Modules**) and contain execution instructions for one or more of imported tools (bundled execution). For instance, a BPH script can call the UPX tool to unpack a compressed UPX executable and the next instruction can call another tool, such as ExeInfoPe or any other tool selected by the user.

The following picture shows the structure of the **PEiD BPH Script**. This script performs the following actions:

- Import required libraries
- Generate a session
- Set the malware sample
- Initialize Template server
- Initialize PEiD's object
- Set scan type (deep_scan) and execute the tool inside the target virtual machine
- Show the tool's output in the console
- List the tool's log file in the console



```python
# Tool imports
from bph.tools.windows.peid import BphPeid as Peid

# Core Imports
from bph.core.server.template import BphTemplateServer as TemplateServer
from bph.core.session import BphSession as Session
from bph.core.sample import BphLabFile as LabFile

session = Session(project_name='blackhat_arsenal_2019')
session.start()
session.set_launcher(move_sample=False)

templateserver = TemplateServer()
templateserver.start()

sample_file = LabFile(session.launcher_abs_path)

peid = Peid(sample_file)
peid.deep_scan()
peid.execute()
peid.output()
peid.files()
```

What is a BPH Analysis Module?

BPH Analysis Modules are python scripts that can be imported and used to perform further analysis of the data collected by **BPH Scripts**. Users can also create code snippets "on-the-fly" within the BPH Script.

BPH Script Development

A BPH script requires a **BPH Module** and a **BPH Plug-In** to be able to run. A BPH Module is the actual tool's **Python class** and the BPH Plug-In is the tool's **(JSON) configuration file**. The following information shows the description of each name/value pairs used for a fictitious application called "MyTool".

Tool Directory Structure & Location

BLACKPHENIX automates malware analysis tasks through the execution of tools. Those tools should be placed into a specific location in the **BPH Controller** (Linux) machine.

Any new tool introduced into the framework must include a BPH Module, Plug-In and the tool itself.

- **BPH Module**
  - ~/bph-framework/bph/tools/windows/mytool.py
- **BPH Plug-In**
  - ~/bph-framework/plugins/basic/static/mytool/x86/1.0/mytool.json
- **Tool (Program & related files)**
  - ~/bph-framework/plugins/basic/static/mytool/x86/1.0/mytool.exe
  - ~/bph-framework/plugins/basic/static/mytool/x86/1.0/xyz.dll

BPH Scripts are not required to have on a specific location. However, it is suggested to use "**~/bph-framework/scripts/**" for such purpose.

NOTE: BLACKPHENIX relies on the folder structure to find BPH related files and tool's program files that follow the convention shown above to ensure proper functioning.

BPH Plug-In (JSON Configuration File)

Any tool used in BLACKPHENIX must have a **BPH Plug-In** configuration file. This file contains the execution data that the tool will use when running on a virtual machine.

```
"template": {
        "name": "MyTool",
        "description": "This is an awesome tool!!"
    }
```

- **name**: The tool's descriptive name
- **description**: The tool's template information

```
 "tool": {
        "filename": "mytool.exe",
        "version": "1.0",
        "type": "gui"
    }
```

- **filename**: The tool's (program) name. Must match with the executable file name
- **version**: The tool's version. Must match with the tool's version
- **type:** The tool's type. It can be either "console" or "gui"

```
"configuration": {
        "execution": {
            "admin_required": false,
            "background_run": false
        },
        "reporting": {
            "report_files": true
        }
   }
```

- **admin_required**: As its name implies
- **background_run**: Whether the tool must run in the background or not. It is running in the background it allows other tools to run while this one is still in execution (e.g. ProcMon, CaptureBAT)
- **report_files**: If set, the tool will report all the generated files to the C&C server. For instance, PEStudio generates XML files, therefore this option is set to "true"

```
"actions": {
        "action_one": {
            "description": "",
            "parameters": "",
            "automation": ""
        },
        "action_two": {
            "description": "",
            "parameters": "",
            "automation": ""
        }
    }
```

- **actions**: A BPH Plug-In can have one or more actions, depending on the analyst's needs
    - For this particular example, two actions are shown: action_one and action_two respectively
- **action_one**: An arbitrary action within an application
- **description**: As its name implies
- **parameters**: The string parameters OR special-variables (@sample@) that are passed as arguments to the tool before its execution. Plain tool arguments can be provided in combination with special variables (i.e.
- **automation**: If the tool requires automation, such as GUI tools, then this field must include an AutoIT script in base64 encoding. The AutoIT script base64 encoded data will be decoded and converted into a temporary script that allows the windows agent running on the virtual machine to run it locally

BPH Plug-In (Special Variables)

Special variables are used to indicate a value that is not currently known but will be generated during the tool's execution. For instance, the special variable "**@sample@**" indicates that once the tool is executed, it must pass as parameter the real sample-path location in the virtual machine. Since all samples are generated using random file names, the **BPH Windows Agent** converts the special variable to the sample's location allowing the tool to work properly.

BPH Special Variables

Variable Name	Description
@sample@	• Sample location
@sample_filename@	• Sample filename
@temp@	• Temporary folder
@appdata@	• OS Application data folder
@tool_drive@	• As its name implies
@tool_name@	• As its name implies
@tool_abs_path@	• Tool's absolute path
@report_folder@	• Tool's report folder

BPH special variables can also be included within AutoIT scripts. There are cases where a dynamically generated value is required, then a special variable will overcome such a problem. This and more special variables can be found on the BPH Windows Agent script **(~/bph-framework/agent/windows/agent.py**).

It is highly suggested to take a look at all the working examples **(~/bph-framework/scripts/examples**) to get a precise idea about how to use special variables.

BPH Plug-In (Parameters)

When a tool gets executed on a virtual machine, it generates output regardless of if type (GUI or console). For instance, Floss program parameters value is set to "@report_folder@\\floss.log", where "floss.log" is the console tool's output.

Many tools also generate additional files, such as Process Monitor, which can save tool's data in PML or generate a report in XML format. All files must be saved locally to be later transmitted to the **BPH Controller** machine (C&C server) and those files will be generated – and controlled – by BPH Plug-In parameters. For instance, SysInspector parameters value is set to: /gen=@report_folder@\\files\\old.xml /privacy /silent, where "old.xml" is the newly generated file.

Plug-In parameters are the actual command-line values passed to a tool before execution. Special variables can be combined with any parameter used by a tool. For instance, the following table shows different BPH Plug-In configuration files and how they handle command-line parameters and values.

nircmd.json:	• "parameters": "killprocess @program@"
autoruns.json	• "parameters":"bdeiklmnoprstw -ct -h -t -o @report_folder@\\files\\autoruns.csv -nobanner"
sysinspector.json	• "parameters": /gen=@report_folder@\\files\\old.xml /privacy /silent • "parameters": "@report_folder@\\files\\old.xml @report_folder@\\files\new.xml"
capturebat.json	• "parameters": "-c -n -l @report_folder@\\files\\capturebat.txt"
pd.json	• "parameters": "-p @process_name@ -o @report_folder@\\files\\"
procmon.json	• "parameters": "/Quiet /AcceptEula /BackingFile @report_folder@\\files\\procmon.pml /LoadConfig @tool_path@\\noisereduction.pmc"
pestudio.json	• "parameters": "-file:@sample@ -xml:@appdata@report.xml && move @appdata@report.xml @report_folder@\\files\\"

depends.json	• "parameters": "/c /oc @report_folder@\\files\\depends.csv @sample@"
floss.json	• "parameters": "@sample@ > @report_folder@\\floss.log"
resourcehacker.json	• "parameters": "-open @sample@ -save @report_folder@\\files\\ -action extract -mask ICONGROUP,, -log @report_folder@\\reshacker.log"
xorstrings.json	• "parameters": "-c @sample@ > @report_folder@\\xorstrings.log"
dumppe.json	• "parameters": "-quiet @sample@ > @report_folder@\\dumppe.log"
xorsearch.json	• "parameters": "-i @sample@ www. > @report_folder@\\xorsearch.log"
upx.json	• "parameters": "-d @sample@ -o @report_folder@\\files\\unpacked.exe > @report_folder@\\upx.log"
Networktraffic.json	• "parameters": "/scomma @report_folder@\\files\\networktrafficview.csv /LoadConfig @tool_path@\\NetworkTrafficView.cfg /CaptureTime 0"

BPH Module (Python Class)

As mentioned earlier, the BPH Module is basically the tool's Class and will attempt to recreate the tool's structure in code to be controlled programmatically by a BPH Script.

BLACKPHENIX uses **"~/bph-framework/bph/tools/windows/"** as the main location for all the tool's BPH modules. When a new BPH Module is developed, the file must be placed there.

```
from termcolor import colored
from bph.core.template import BphToolTemplateExecutor
```

- **Imports**:
    - **colored**: Provide colored console output
- **BphToolTemplateExecutor**: Template Execution code

```
class BphMyTool(BphToolTemplateExecutor):
  def __init__(self, target_file=None, tool_name='mytool', arch='x86', version='1.0'):
        super().__init__()
        self.load_tool_config_file(tool_name, arch, version, target_file=target_file)
```

- **Class Name** (BphMyTool): Must use "BPH" before its name. The name can be the tool's name as convention
    - Must inherit from BphToolTemplateExecutor
- **Init method:**
    - **tool_name**: As its name implies
    - **arch:** The tools' architecture
    - **version**: As its name implies
- **self.load_tool_config_file** and **super()** methods inclusion, must not be modified

```
    def action_one(self):
        self.logger.log(colored('EXECUTING ACTION ONE', 'yellow'))
        self.actions.action = "action_one"
```

- **action_one method:** One or more methods can be created. The method "action_one" must map to the actions set in the JSON configuration file.
- **logger.log method**: Shows output in the console
- The values within the JSON file can also be dynamically modified from the BPH Module. In this case "self.actions.action" sets the JSON value to "action_one" respectively.

BPH Script (Programmatic Execution)

Once the tool's BPH Plug-In and Module are done, then a BPH Script should be developed. The following table contains the code and explanation about how it is composed.

```
from bph.tools.windows.mytool import BphMyTool as MyTool
```

- **Imports**:
    - For easy access, is suggested to use the tool's name

```
from bph.core.server.template import BphTemplateServer as TemplateServer
from bph.core.session import BphSession as Session
from bph.core.sample import BphLabFile as LabFile
```

- **Imports**:
    - BphTemplateServer, BphSession, and BphLabFile must be imported to ensure the proper functioning

```
session = Session(project_name='mysession')
session.start()
session.set_launcher(move_sample=False)
```

- **Session() – project_name:** String used as a name for the project
- **session.start():** Initialize the BPH session
- **session.set_launcher()**: Initialize the malware sample to use during the session
    - **move_sample:** To move the sample from the temporary location to the BLACKPHENIX sample's folder

```
templateserver = TemplateServer()
templateserver.start()
```

- **TemplateServer():** Instantiate the Template server (C&C)
    - Used for network communication between the Windows BPH Analysis (Guest VM) and the BPH Controller (Linux VM)
- **start()**: Initialize the Template server

```
sample_file = LabFile(session.launcher_abs_path)
```

- **LabFile()**: Instantiate the malware sample to use in the current session
- **sample_file:** The sample's objects attributes can be also accessed (e.g. sample_file.md5, sample_file.abs_path)

```
mytool = MyTool(sample_file)
mytool.action_one()
mytool.execute()
mytool.output()
mytool.files()
```

- **MyTool():** Instantiate the tool by passing the sample file object
- **mytool.action_one():** This method is selected to be executed by the tool
- **mytool.execute():** This method instructs the Template Server to perform the actual delivery of the JSON configuration file to the Windows Agent running on the (BPH Analysis) virtual machine
- **mytool.output():** Once the execution is done, this method call shows the tool's output in the console
- **mytool.files():** This method shows the collected files from the tool's execution
- Both (**output()**, and **files()**) resulting output can also be placed inside or a variable for further processing

BPH Analysis Module Development

BPH Analysis Modules are used to consume the data collected by the tools, or in other words, the results given by the BPH scripts. BLACKPHENIX uses "**~/bph-framework/bph/analysis/**" as the main location for all the tool's BPH Analysis modules. When a new BPH Analysis Module is developed, the file must be placed there.

For a better understanding of a BPH Analysis module development, a current framework's module (NetworkTrafficView) will be used and will focus on the Analysis Module key parts.

```
from bph.analysis.network import BphNetworkAnalysisCsvReader as NetworkAnalysisCsvRead
er
```

- **Imports**:
  - NetworkAnalysisCsvReader

```
[…]
ntv = NetworkTrafficView()
ntv.start()
ntv.execute()
[…]
ntv.stop()
ntv.execute()

for csv_file in ntv.files():
    ntv = NetworkAnalysisCsvReader(tool_name='networktrafficview', csv_file=csv_file)
    ntv.fetch(data_type='domains')
```

- **ntv.files():** Iterate in the tool's collected files (absolute path) and stored in a variable (csv_file)
- **NetworkAnalysisCsvReaser():** Instantiate the tool's object and pass the "csv_file" variable which contains the absolute path of the collected file
- **ntv.fetch()**: Filter all domain names collected from the network traffic

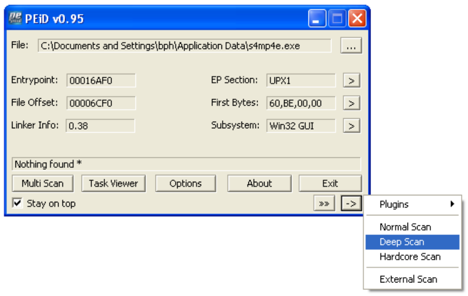
The framework examples include working scripts for the NetworkTrafficView tool:

- **BPH Module**: **~/bph-framework/bph/tools/windows/networktrafficview.py**

- **BPH Analysis Module: ~/bph-framework/bph/analysis/network.py**
- **BPH Script: ~/bph-framework/scripts/examples/tools/networktrafficview_python.py**

PEiD Walkthrough – GUI-based Application

PEID is a well-known tool used for packer identification. The tool provides different “scanning” options: **Normal Scan**, **Deep Scan**, **Hardcore Scan,** and **External Scan** respectively.



PEiD v0.95

File: C:\Documents and Settings\bph\Application Data\s4mp4e.exe

Entrypoint: 00016AF0 | EP Section: UPX1
File Offset: 00006CF0 | First Bytes: 60,BE,00,00
Linker Info: 0.38 | Subsystem: Win32 GUI

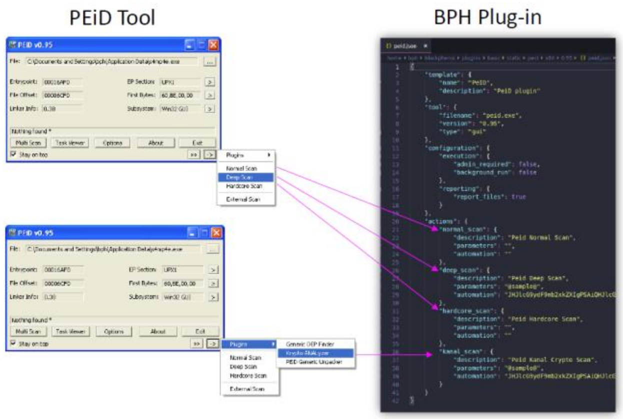
Nothing found *

These options are available for the user to use and provide different “scan” options. From the BLACKPHENIX perspective, these options can be automatically selected, just as the user does when using the tool manually.

PEiD – BPH Plug-in

The first step to automate PEiD is to create its BPH Plug-In, or in other words, its JSON configuration file. Each tool option (the scan features) must have its related name/value data in the configuration file.

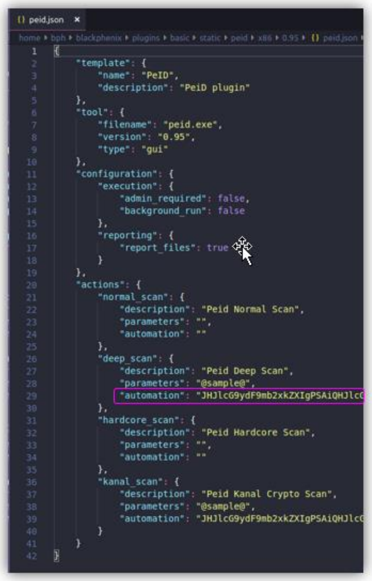
The following picture shows the relationship between the configuration file and the tool features.



```
{
    "template": {
        "name": "PeID",
        "description": "PeID plugin"
    },
    "tool": {
        "filename": "peid.exe",
        "version": "0.95",
        "type": "gui"
    },
    "configuration": {
        "execution": {
            "admin_required": false,
            "background_run": false
        },
        "reporting": {
            "report_files": true
        }
    },
    "actions": {
        "normal_scan": {
            "description": "PeiD Normal Scan",
            "parameters": "",
            "automation": ""
        },
        "deep_scan": {
            "description": "PeiD Deep Scan",
            "parameters": "@sample@",
            "automation": "JHJlcG9ydF9mb2xkZXIgPSAiQHJlc
        },
        "hardcore_scan": {
            "description": "PeiD Hardcore Scan",
            "parameters": "",
            "automation": ""
        },
        "kanal_scan": {
            "description": "PeiD Kanal Crypto Scan",
            "parameters": "@sample@",
            "automation": "JHJlcG9ydF9mb2xkZXIgPSAiQHJlc
        }
    }
}
```

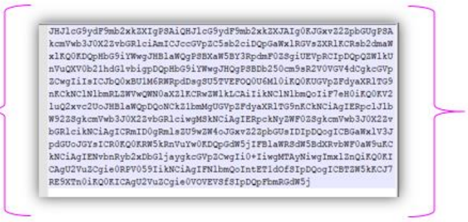
The deep_scan “parameters” name shows “@sample@”, which will be converted to the sample’s path when the tool is executed on the virtual machine. The automation value contains base64 encoded data that will be decoded and executed by AutoIT when running on the virtual machine after the tool gets executed.

The following picture shows the AutoIt script content and how is used as an automation value for the deep_scan option.

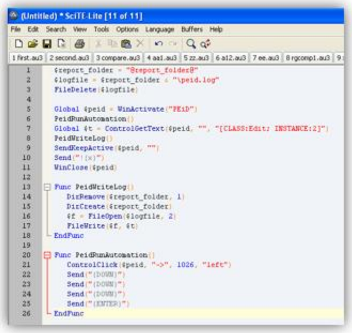


peid.json
home › bph › blackphenix › plugins › basic › static › peid › x86 › 0.95 › peid.json

```json
{
    "template": {
        "name": "PeID",
        "description": "PeiD plugin"
    },
    "tool": {
        "filename": "peid.exe",
        "version": "0.95",
        "type": "gui"
    },
    "configuration": {
        "execution": {
            "admin_required": false,
            "background_run": false
        },
        "reporting": {
            "report_files": true
        }
    },
    "actions": {
        "normal_scan": {
            "description": "Peid Normal Scan",
            "parameters": "",
            "automation": ""
        },
        "deep_scan": {
            "description": "Peid Deep Scan",
            "parameters": "@sample@",
            "automation": "JHJlcG9ydF9mb2xkZXIgPSAiQHJlcG
        },
        "hardcore_scan": {
            "description": "Peid Hardcore Scan",
            "parameters": "",
            "automation": ""
        },
        "kanal_scan": {
            "description": "Peid Kanal Crypto Scan",
            "parameters": "@sample@",
            "automation": "JHJlcG9ydF9mb2xkZXIgPSAiQHJlcG
        }
    }
}
```



```
JHJlcG9ydF9mb2xkZXIgPSAiQHJlcG9ydF9mb2xkZXJAIg0KJGxvZ2ZpbGUgPSA
kcmVwb3J0X2ZvbGRlciAmICJccGVpZC5sb2ciDQpGaWxlRGVsZXRlKCRsb2dmaW
xlKQ0KDQpHbG9iYWwgJHBlaWQgPSBXaW5BY3RpdmF0ZSgiUEVpRCIpDQpQZWlkU
nVuQXV0b21hdGlvbigpDQpHbG9iYWwgJHQgPSBDb250cm9sR2V0VGV4dCgkcGVp
ZCwgIiIsICJbQ0xBU1M6RWRpdDsgSU5TVEFOQ0U6Ml0iKQ0KUGVpZFdyaXRlTG9
nKCkNClNlbmRLZWVwQWN0aXZlKCRwZWlkLCAiIikNClNlbmQoIiF7eH0iKQ0KV2
luQ2xvc2UoJHBlaWQpDQoNCkZ1bmMgUGVpZFdyaXRlTG9nKCkNCiAgIERpclJlb
W92ZSgkcmVwb3J0X2ZvbGRlciwgMSkNCiAgIERpckNyZWF0ZSgkcmVwb3J0X2Zv
bGRlcikNCiAgICRmID0gRmlsZU9wZW4oJGxvZ2ZpbGUsIDIpDQogICBGaWxlV3J
pdGUoJGYsICR0KQ0KRW5kRnVuYw0KDQpGdW5jIFBlaWRSdW5BdXRvbWF0aW9uKC
kNCiAgIENvbnRyb2xDbGljaygkcGVpZCwgIi0+IiwgMTAyNiwgImxlZnQiKQ0KI
CAgU2VuZCgie0RPV059IikNCiAgIFNlbmQoIntET1dOfSIpDQogICBTZW5kKCJ7
RE9XTn0iKQ0KICAgU2VuZCgie0VOVEVSfSIpDQpFbmRGdW5j
```

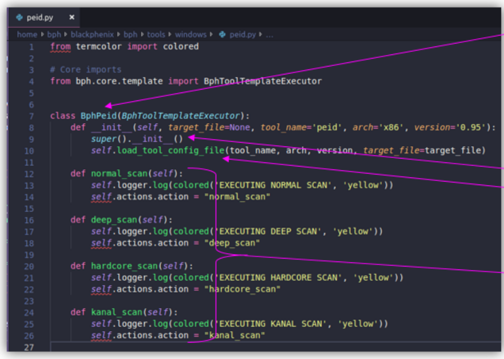


(Untitled) * SciTE-Lite [11 of 11]
File Edit Search View Tools Options Language Buffers Help
1 first.au3
2 second.au3
3 compare.au3
4 aa1.au3
5 zz.au3
6 a12.au3
7 ee.au3
8 rgcomp1.au3

```autoit
$report_folder = "@report_folder@"
$logfile = $report_folder & "\peid.log"
FileDelete($logfile)

Global $peid = WinActivate("PEiD")
PeidRunAutomation()
Global $t = ControlGetText($peid, "", "[CLASS:Edit; INSTANCE:2]")
PeidWriteLog()
SendKeepActive($peid, "")
Send("!{x}")
WinClose($peid)

Func PeidWriteLog()
   DirRemove($report_folder, 1)
   DirCreate($report_folder)
   $f = FileOpen($logfile, 2)
   FileWrite($f, $t)
EndFunc

Func PeidRunAutomation()
   ControlClick($peid, "->", 1026, "left")
   Send("{DOWN}")
   Send("{DOWN}")
   Send("{DOWN}")
   Send("{ENTER}")
EndFunc
```

It is understood that the AutoIt script should contain the corresponding automation to select the "Deep Scan" menu/option once PEiD is running. The HOW-TO guide about how to write AutoIt scripts goes beyond this guide. However, a BPH developer can get further information from the AutoIt website.

PEiD – BPH Module

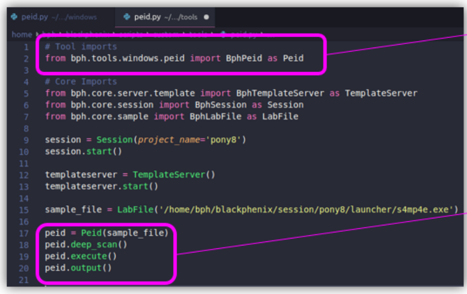


peid.py
home › bph › blackphenix › bph › tools › windows › peid.py › ...

```python
from termcolor import colored

# Core imports
from bph.core.template import BphToolTemplateExecutor


class BphPeid(BphToolTemplateExecutor):
    def __init__(self, target_file=None, tool_name='peid', arch='x86', version='0.95'):
        super().__init__()
        self.load_tool_config_file(tool_name, arch, version, target_file=target_file)

    def normal_scan(self):
        self.logger.log(colored('EXECUTING NORMAL SCAN', 'yellow'))
        self.actions.action = "normal_scan"

    def deep_scan(self):
        self.logger.log(colored('EXECUTING DEEP SCAN', 'yellow'))
        self.actions.action = "deep_scan"

    def hardcore_scan(self):
        self.logger.log(colored('EXECUTING HARDCORE SCAN', 'yellow'))
        self.actions.action = "hardcore_scan"

    def kanal_scan(self):
        self.logger.log(colored('EXECUTING KANAL SCAN', 'yellow'))
        self.actions.action = "kanal_scan"
```

- **BPH Module == Tool's Python Class**
  - A Class should be created: *BphMyTool()*
    - This class inherits from *BphToolTemplateExecutor* class, which is a class that acts as interface between BPH plug-ins, scripts a Virtual Machine Network Agent
  - Initialization function ("*__init__*" method) is in charge of:
    - Initializing the class with tool's name, architecture, and version number
    - Calling *super()* to inherit methods and attributes from its parent class
    - Loading BPH-Plug-In JSON configuration file by calling *self.load_tool_config_file()* method and tool's data as parameters
  - For each Tool's feature, a class method will be used
    - Each method must enable the action that will be called/executed
    - It modifies values inside the JSON (BPH Plug-In) configuration data
  - **In BLACKPHENIX, *all* tools are Python objects (Classes) with its correspondent attributes!**

PEiD – BPH Script



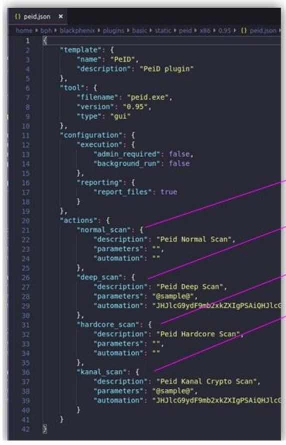
```
# Tool imports
from bph.tools.windows.peid import BphPeid as Peid

# Core Imports
from bph.core.server.template import BphTemplateServer as TemplateServer
from bph.core.session import BphSession as Session
from bph.core.sample import BphLabFile as LabFile

session = Session(project_name='pony8')
session.start()

templateserver = TemplateServer()
templateserver.start()

sample_file = LabFile('/home/bph/blackphenix/session/pony8/launcher/s4mp4e.exe')

peid = Peid(sample_file)
peid.deep_scan()
peid.execute()
peid.output()
```

- **BPH Script == Execution Instructions**
  - The tool's Class should be imported
    - *From bph.tools.<os>.<tool_name> import BphMyTool as MyTool*
  - The tool's object can be created as a regular Python object
    - Some tools require initialization parameters, especially the ones that work with samples or files
    - Some other tools (such as Process Monitor or CaptureBAT are not tied to a sample, but collecting data from a virtualized sample execution
  - After object's creation, then its methods can be called
    - Tools that run inside the Virtual Machine requires a call to the execute() method
    - BPH uses two main methods: output() and files()
      - *Output()* returns the tool's output
      - *Files()* returns the folder location where the collected files resides on
  - On a single BPH Scripts a user can choose a single tool or a bundle to perform different actions and use the data generated by all of them!

PEiD – All together

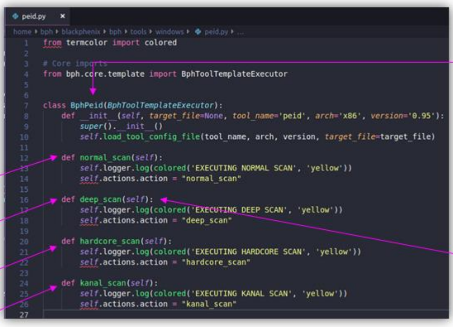
For better understanding, the following picture shows the graphical relationship between these three components.

BPH Plug-in



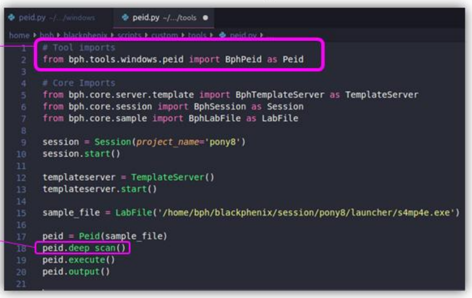
```
{
    "template": {
        "name": "PeID",
        "description": "PeiD plugin"
    },
    "tool": {
        "filename": "peid.exe",
        "version": "0.95",
        "type": "gui"
    },
    "configuration": {
        "execution": {
            "admin_required": false,
            "background_run": false
        },
        "reporting": {
            "report_files": true
        }
    },
    "actions": {
        "normal_scan": {
            "description": "Peid Normal Scan",
            "parameters": "",
            "automation": ""
        },
        "deep_scan": {
            "description": "Peid Deep Scan",
            "parameters": "@sample@",
            "automation": "JHJlcG9ydF9mb2xkZXIgPSAiQHJlcG
        },
        "hardcore_scan": {
            "description": "Peid Hardcore Scan",
            "parameters": "",
            "automation": ""
        },
        "kanal_scan": {
            "description": "Peid Kanal Crypto Scan",
            "parameters": "@sample@",
            "automation": "JHJlcG9ydF9mb2xkZXIgPSAiQHJlcG
        }
    }
}
```

BPH Module



```
from termcolor import colored

# Core imports
from bph.core.template import BphToolTemplateExecutor


class BphPeid(BphToolTemplateExecutor):
    def __init__(self, target_file=None, tool_name='peid', arch='x86', version='0.95'):
        super().__init__()
        self.load_tool_config_file(tool_name, arch, version, target_file=target_file)

    def normal_scan(self):
        self.logger.log(colored('EXECUTING NORMAL SCAN', 'yellow'))
        self.actions.action = "normal_scan"

    def deep_scan(self):
        self.logger.log(colored('EXECUTING DEEP SCAN', 'yellow'))
        self.actions.action = "deep_scan"

    def hardcore_scan(self):
        self.logger.log(colored('EXECUTING HARDCORE SCAN', 'yellow'))
        self.actions.action = "hardcore_scan"

    def kanal_scan(self):
        self.logger.log(colored('EXECUTING KANAL SCAN', 'yellow'))
        self.actions.action = "kanal_scan"
```

BPH Script



```
# Tool imports
from bph.tools.windows.peid import BphPeid as Peid

# Core Imports
from bph.core.server.template import BphTemplateServer as TemplateServer
from bph.core.session import BphSession as Session
from bph.core.sample import BphLabFile as LabFile

session = Session(project_name='pony8')
session.start()

templateserver = TemplateServer()
templateserver.start()

sample_file = LabFile('/home/bph/blackphenix/session/pony8/launcher/s4mp4e.exe')

peid = Peid(sample_file)
peid.deep_scan()
peid.execute()
peid.output()
```

Contact

For general information regarding the framework, please use the following email

- bph_framework@fortinet.com