```python
#!/usr/bin/python3
#
# This software is provided under under the BSD 3-Clause License.
# See the accompanying LICENSE file for more information.
#
# Windows Exploit Suggester - Next Generation
#
# Author: Arris Huijgen (@bitsadmin)
# Website: https://github.com/bitsadmin
'''Requirements: chardet; platform_system == "Windows" '''

from __future__ import print_function

import sys, csv, re, argparse, os, zipfile, io
import logging
from collections import Counter, OrderedDict
import copy
import tempfile

# Python 2 compatibility
if sys.version_info.major == 2:
    from urllib import urlretrieve

    ModuleNotFoundError = ImportError
else:
    from urllib.request import urlretrieve

# Check availability of the chardet library:
# "The universal character encoding detector"
try:
    import chardet


    # Using chardet library to determine the approperiate encoding
    def charset_convert(data):
        encoding = chardet.detect(data)
        data = data.decode(encoding['encoding'], 'ignore')

        if sys.version_info.major == 2:
            data = data.encode(sys.getfilesystemencoding())

        return data

except (ImportError, ModuleNotFoundError):
    # Parse everything as ASCII
    def charset_convert(data):
        data = data.decode('ascii', 'ignore')

        if sys.version_info.major == 2:
            data = data.encode(sys.getfilesystemencoding())

        return data


    logging.warning(
        'chardet module not installed. In case of encoding '
        'errors, install chardet using: pip{} install chardet'.format(sys.version_info
        .major))


# By default show plain output without color
def colored(text, color):
    return text
```

```
72
73  def configure_color():
74      # Check availability of the termcolor library
75      try:
76          global colored
77          from termcolor import colored
78
79      except (ImportError, ModuleNotFoundError):
80          logging.warning('termcolor module not installed. To show colored output, '
81                          'install termcolor using: pip{} install termcolor'.format(sys.
                            version_info.major))
82          pass
83
84      # Also check availability of the colorama library in case of Windows
85      try:
86          if os.name == 'nt':
87              import colorama
88              colorama.init()
89      except (ImportError, ModuleNotFoundError):
90          logging.warning('colorama module not installed. To show colored output in
            Windows, '
91                          'install colorama using: pip{} install colorama'.format(sys.
                            version_info.major))
92          pass
93
94
95  class WesException(Exception):
96      pass
97
98
99  # Application details
100 TITLE = 'Windows Exploit Suggester'
101 VERSION = 1.03
102 RELEASE = ''
103 WEB_URL = 'https://github.com/bitsadmin/wesng/'
104 BANNER = '%s %s ( %s )'
105 FILENAME = 'wes.py'
106
107 # Mapping table between build numbers and versions to correctly identify
108 # the Windows 10/11/Server 2016/2019/2022 version specified in the systeminfo output
109
110 buildnumbers = OrderedDict([
111     (10240, 1507),
112     (10586, 1511),
113     (14393, 1607),
114     (15063, 1703),
115     (16299, 1709),
116     (17134, 1803),
117     (17763, 1809),
118     (18362, 1903),
119     (18363, 1909),
120     (19041, 2004),
121     (19042, '20H2'),
122     (19043, '21H1'),
123     (19044, '21H2'), # Windows 10
124     (19045, '22H2'),
125     (20348, '21H2'), # Windows Server 2022
126     (22000, '21H2'),  # Windows 11
127     (22621, '22H2')
128 ])
129
130
131
132
133
134
135
136
137
138
139
140
```

```
141
142 def main(args, tmp_dir):
143     # Configure output coloring
144     if args['showcolor']:
145         configure_color()
146
147     # Application banner
148     print(BANNER % (colored(TITLE, 'green'), colored('%.2f' % VERSION, 'yellow'),
        colored(WEB_URL, 'blue')))
149
150     # Update definitions
151     if args['perform_update']:
152         print(colored('[+] Updating definitions', 'green'))
153         urlretrieve(
            'https://raw.githubusercontent.com/bitsadmin/wesng/master/definitions.zip',
154                     f'{tmp_dir}/definitions.zip')
155         cves, date = load_definitions(f'{tmp_dir}/definitions.zip')
156         print(colored('[+] Obtained definitions created at ', 'green') + '%s' %
            colored(date, 'yellow'))
157
158     # Show tree of supersedes (for debugging purposes)
159     if args['debugsupersedes']:
160         cves, date = load_definitions(f'{tmp_dir}/definitions.zip')
161         productfilter = args['debugsupersedes'][0]
162         supersedes = args['debugsupersedes'][1:]
163         filtered = []
164         for cve in cves:
165             if productfilter not in cve['AffectedProduct']:
166                 continue
167
168             filtered.append(cve)
169
170         debug_supersedes(filtered, supersedes, 0, args['verbosesupersedes'])
171         return
172
173     # Show version
174     if args['showversion']:
175         cves, date = load_definitions(f'{tmp_dir}/definitions.zip')
176         print('Wes.py version: %.2f' % VERSION)
177         print('Database version: %s' % date)
178
179     # Using the list of missing patches as a base
180     if args['missingpatches']:
181         print(colored('[+] Loading definitions', 'green'))
182         cves, date = load_definitions(f'{tmp_dir}/definitions.zip')
183
184         # Obtain IDs of missing patches from file
185         print(colored('[+] Loading missing patches from file', 'green'))
186         missingpatches = []
187         with open(args['missingpatches'], 'r') as f:
188             missingpatches = f.read()
189         missingpatches = list(filter(None, [mp.upper().replace('KB', '') for mp in
            missingpatches.splitlines()]))
190
191         # Obtain all records matching the IDs of the missing patches
192         found = list(filter(lambda c: c['BulletinKB'] in missingpatches, cves))
193         os_names, os_name = get_operatingsystems(found, args['operating_system'])
194
195         # Perform filter on operating system
196         if os_name:
197             print(colored('[+] Filtering vulnerabilities for "%s"' % os_name, 'green'
                ))
198             found = list(filter(lambda c: os_name in c['AffectedProduct'], found))
199
200
201
202
203
204
205
206
207
```

```
208
209             # Deduplicate results ignoring differences in the Supersedes attribute
210             for f in found:
211                 f['Supersedes'] = ''
212             found = [dict(t) for t in {tuple([t for t in d.items()]) for d in found}]
213
214             # Append missing patches from missing.txt which are not included in the
                definitions.zip
215             foundkbs = set([kb['BulletinKB'] for kb in found])
216             difference = foundkbs.symmetric_difference(missingpatches)
217             for diff in difference:
218                 found.append({'DatePosted': '', 'CVE': '', 'BulletinKB': diff, 'Title': ''
                    , 'AffectedProduct': '',
219                               'AffectedComponent': '', 'Severity': '', 'Impact': '',
                              'Supersedes': '', 'Exploits': ''})
220
221             if os_name and 'Windows Server' in os_name:
222                 print(colored('[+] Filtering duplicate vulnerabilities', 'green'))
223                 found = filter_duplicates(found)
224
225             # Prepare variables for summary
226             sp = None
227             kbs = found
228
229         # Using systeminfo.txt or qfe.txt with list of installed patches as a base
230         else:
231             missingpatches = None
232             cves = None
233             os_names = None
234             os_name = None
235
236             # Use input from qfe
237             if args['qfefile']:
238                 # If an operating_system digit is provided or no OS has been provided,
                    load defitions to
239                 # respectively retrieve the OS or show the list of OSs
240                 if (args['operating_system'] and args['operating_system'].isdigit()) or \
241                         (not args['operating_system']):
242                     # Load definitions to compile list of OSs
243                     print(colored('[+] Loading definitions', 'green'))
244                     cves, date = load_definitions(f'{tmp_dir}/definitions.zip')
245                     print('    - Creation date of definitions: %s' % date)
246
247                 # Propose/select OS name
248                 os_names, os_name = get_operatingsystems(cves, args['operating_system'])
249                 if not args['operating_system']:
250                     # Print possible operating systems
251                     list_operatingsystems(os_names)
252
253                     # Quit script
254                     print(colored(
255                         '[I] Rerun the script providing the --os parameter and the index
                            or name of the OS you want to filter on.',
256                         'yellow'))
257                     exit(0)
258                 else:
259                     productfilter = os_name
260
261                 # Read KBs from QFE file
262                 print(colored('[+] Parsing quick fix engineering (qfe) output', 'green'))
263                 with open(args['qfefile'], 'rb') as f:
264                     qfe_data = f.read()
265                 qfe_data = charset_convert(qfe_data)
266                 hotfixes = get_hotfixes(qfe_data)
267
268
269
270
271
272
273
274
```

```
275
276         # Parse encoding of systeminfo.txt input
277         else:
278             print(colored('[+] Parsing systeminfo output', 'green'))
279             systeminfo_data = open(args['systeminfo'], 'rb').read()
280             try:
281                 productfilter, win, mybuild, version, arch, hotfixes =
                    determine_product(systeminfo_data)
282             except WesException as e:
283                 print(colored('[-] ' + str(e), 'red'))
284                 exit(1)
285
286         # Add explicitly specified patches
287         manual_hotfixes = list(set([patch.upper().replace('KB', '') for patch in args[
            'installedpatch']]))
288
289         # Display summary
290         # OS info
291         info = colored('[+] Operating System', 'green')
292         if args['systeminfo']:
293             info += ('\n'
294                      '    - Name: %s\n'
295                      '    - Generation: %s\n'
296                      '    - Build: %s\n'
297                      '    - Version: %s\n'
298                      '    - Architecture: %s') % (productfilter, win, mybuild, version
                         , arch)
299         elif os_name:
300             info += '\n    - Selected Operating System: %s' % os_name
301
302         # Hotfixes
303         if hotfixes:
304             info += '\n    - Installed hotfixes (%d): %s' % (len(hotfixes), ', '.join
                (['KB%s' % kb for kb in hotfixes]))
305         else:
306             info += '\n    - Installed hotfixes: None'
307         if manual_hotfixes:
308             info += '\n    - Manually specified hotfixes (%d): %s' % (len(
                manual_hotfixes),
309                                                                       ', '.join(
310                                                                           ['KB%s' % kb
    for kb in manual_hotfixes]))
311         print(info)
312
313         # Append manually specified KBs to list of hotfixes
314         hotfixes = list(set(hotfixes + manual_hotfixes))
315         hotfixes_orig = copy.deepcopy(hotfixes)
316
317         # Load definitions from definitions.zip (default) or user-provided location
318         # Only in case they haven't been loaded yet when the --qfe parameter has been
            provided
319         if not cves:
320             print(colored('[+] Loading definitions', 'green'))
321             cves, date = load_definitions(f'{tmp_dir}/definitions.zip')
322             print('    - Creation date of definitions: %s' % date)
323
324         # Determine missing patches
325         try:
326             print(colored('[+] Determining missing patches', 'green'))
327             filtered, found = determine_missing_patches(productfilter, cves, hotfixes)
328         except WesException as e:
329             print(colored('[-] ' + str(e), 'red'))
330             exit(1)
331
332
333
334
335
336
337
338
339
```

```
340
341         # If -d parameter is specified, use the most recent patch installed as
342         # reference point for the system's patching status
343         if args['usekbdate']:
344             print(colored('[+] Filtering old vulnerabilities', 'green'))
345             recentkb = get_most_recent_kb(found)
346             if recentkb:
347                 print('    - Most recent KB installed is KB%s released at %s\n'
348                       '    - Filtering all KBs released before this date' % (
349                       recentkb['BulletinKB'], recentkb['DatePosted']))
350                 recentdate = int(recentkb['DatePosted'])
351                 found = list(filter(lambda kb: int(kb['DatePosted']) >= recentdate,
                    found))
352
353         if 'Windows Server' in productfilter:
354             print(colored('[+] Filtering duplicate vulnerabilities', 'green'))
355             found = filter_duplicates(found)
356
357     # If specified, hide results containing the user-specified string
358     # in the AffectedComponent and AffectedProduct attributes
359     if args['hiddenvuln'] or args['only_exploits'] or args['impacts'] or args[
    'severities']:
360         print(colored('[+] Applying display filters', 'green'))
361         filtered = apply_display_filters(found, args['hiddenvuln'], args[
            'only_exploits'], args['impacts'],
362                                          args['severities'])
363     else:
364         filtered = found
365
366     # In case the list of missing patches is specified,
367     # we don't need to search for supersedes in the MS Update Catalog
368     if not args['missingpatches']:
369         # If specified, lookup superseded KBs in the Microsoft Update Catalog
370         # and remove CVEs if a superseded KB is installed.
371         if args['muc_lookup']:
372             from muc_lookup import apply_muc_filter  # ony import if necessary since
                it needs MechanicalSoup
373
374             print(colored('[!] Looking up superseded hotfixes in the Microsoft Update
                Catalog', 'yellow'))
375             filtered = apply_muc_filter(filtered, hotfixes_orig)
376
377         # Split up list of KBs and the potential Service Packs/Cumulative updates
            available
378         kbs, sp = get_patches_servicepacks(filtered, cves, productfilter)
379
380     # Display results
381     if len(filtered) > 0:
382         print(colored('[!] Found vulnerabilities!', 'yellow'))
383         if args['outputfile']:
384             store_results(args['outputfile'], filtered)
385             verb = 'Saved'
386             print_summary(kbs, sp)
387         else:
388             print_results(filtered)
389             verb = 'Displaying'
390             print_summary(kbs, sp)
391
392         if not args['operating_system'] and os_names and len(os_names) > 1:
393             # Print possible operating systems
394             list_operatingsystems(os_names)
395
396             print(colored('[I] Additional filter can be applied using the --os
                parameter', 'yellow'))
397
398         print(colored('[+] Done. ', 'green') + '%s %s of the %s vulnerabilities
            found.' % (
399         verb, colored(len(filtered), 'yellow'), colored(len(found), 'yellow')))
400     else:
401         print(colored('[-] Done. No vulnerabilities found\n', 'green'))
402
403
```

```
404
405     # Load definitions.zip containing a CSV with vulnerabilities collected by the WES
        collector module
406     # and a file determining the minimum wes.py version the definitions are compatible
        with.
407     def load_definitions(definitions):
408         with zipfile.ZipFile(definitions, 'r') as definitionszip:
409             files = definitionszip.namelist()
410
411             # Version_X.XX.txt
412             versions = list(filter(lambda f: f.startswith('Version'), files))
413             versionsfile = versions[0]
414             dbversion = float(re.search('Version_(.*)\.txt', versionsfile, re.MULTILINE |
                re.IGNORECASE).group(1))
415
416             if dbversion > VERSION:
417                 raise WesException(
418                     'Definitions require at least version %.2f of wes.py. '
419                     'Please update using wes.py --update-wes.' % dbversion)
420
421             # CVEs_yyyyMMdd.csv
422             #
                DatePosted,CVE,BulletinKB,Title,AffectedProduct,AffectedComponent,Severity,Imp
                act,Supersedes,Exploits
423             cvesfiles = list(filter(lambda f: f.startswith('CVEs'), files))
424             cvesfile = cvesfiles[0]
425             cvesdate = cvesfile.split('.')[0].split('_')[1]
426             f = io.TextIOWrapper(definitionszip.open(cvesfile, 'r'))
427             cves = csv.DictReader(filter(lambda row: row[0] != '#', f), delimiter=str(','
                ), quotechar=str('"'))
428
429             # Custom_yyyyMMdd.csv
430             customfiles = list(filter(lambda f: f.startswith('Custom'), files))
431             customfile = customfiles[0]
432             f = io.TextIOWrapper(definitionszip.open(customfile, 'r'))
433             custom = csv.DictReader(filter(lambda row: row[0] != '#', f), delimiter=str(
                ','), quotechar=str('"'))
434
435             # Merge official and custom list of CVEs
436             merged = [cve for cve in cves] + [c for c in custom]
437
438             return merged, cvesdate
439
440
441     # Hide results based on filter(s) specified by the user. This can either be to only
        display results with
442     # public exploits, results with a given impact or results containing the user
        specified string(s) in
443     # the AffectedComponent or AffectedProduct attributes.
444     def apply_display_filters(found, hiddenvulns, only_exploits, impacts, severities):
445         # --hide 'Product 1' 'Product 2'
446         hiddenvulns = list(map(lambda s: s.lower(), hiddenvulns))
447         impacts = list(map(lambda s: s.lower(), impacts))
448         severities = list(map(lambda s: s.lower(), severities))
449         filtered = []
450         for cve in found:
451             add = True
452             for hidden in hiddenvulns:
453                 if hidden in cve['AffectedComponent'].lower() or hidden in cve[
                    'AffectedProduct'].lower() or hidden in cve[
454                     'Title'].lower():
455                     add = False
456                     break
457
458
459
460
461
462
463
464
465
```

```
466
467             for impact in impacts:
468                 if not impact in cve['Impact'].lower():
469                     add = False
470                 else:
471                     add = True
472                     break
473
474             for severity in severities:
475                 if not severity in cve['Severity'].lower():
476                     add = False
477                 else:
478                     add = True
479                     break
480
481             if add:
482                 filtered.append(cve)
483
484         # --exploits-only
485         if only_exploits:
486             filtered = list(filter(lambda res: res['Exploits'], filtered))
487
488         return filtered
489
490
491     # Filter duplicate CVEs for the Windows Server operating systems which often have a
492     # 'Windows Server 2XXX' and a 'Windows Server 2XXX (Server Core installation)' CVE
        that are exactly the same
493     def filter_duplicates(found):
494         cves = list(set([cve['CVE'] for cve in found]))
495         newfound = []
496
497         # Iterate over unique CVEs
498         for cve in cves:
499             coreresults = list(filter(lambda cr: cr['CVE'] == cve and 'Server Core' in cr[
                'AffectedProduct'], found))
500
501             # If no 'Server Core' results for CVE, just add all records matching the CVE
502             if len(coreresults) == 0:
503                 normalresults = list(filter(lambda nr: nr['CVE'] == cve, found))
504                 for n in normalresults:
505                     newfound.append(n)
506                 continue
507
508             # In case 'Server Core' records are found, identify matching non-core results
509             for r in coreresults:
510                 regularcounterparts = list(filter(lambda c:
511                                                   'Server Core' not in c['AffectedProduct'
                                                  ] and
512                                                   c['CVE'] == r['CVE'] and
513                                                   c['BulletinKB'] == r['BulletinKB'] and
514                                                   c['Title'] == r['Title'] and
515                                                   c['AffectedComponent'] == r[
                                                      'AffectedComponent'] and
516                                                   c['Severity'] == r['Severity'] and
517                                                   c['Impact'] == r['Impact'] and
518                                                   c['Exploits'] == r['Exploits'], found))
519
520                 # If non-'Server Core' counterparts are found, add these
521                 if len(regularcounterparts) >= 1:
522                     for rc in regularcounterparts:
523                         newfound.append(rc)
524                 # Otherwise, add the 'Server Core' CVE
525                 else:
526                     newfound.append(r)
527
528         return newfound
529
530
531
532
533
```

```
534
535 # Filter CVEs that are applicable to this system
536 def determine_missing_patches(productfilter, cves, hotfixes):
537     filtered = []
538
539     # Product with a Service Pack
540     if 'Service Pack' in productfilter:
541         for cve in cves:
542             if productfilter not in cve['AffectedProduct']:
543                 continue
544
545             cve['Relevant'] = True
546             filtered.append(cve)
547
548             if cve['Supersedes']:
549                 hotfixes.append(cve['Supersedes'])
550     # Make sure that if the productfilter does not contain a Service Pack, we don't
        list the versions of that OS
551     # which include a Service Pack in the product name
552     else:
553         productfilter_sp = productfilter + ' Service Pack'
554         for cve in cves:
555             if productfilter not in cve['AffectedProduct'] or productfilter_sp in cve[
                'AffectedProduct']:
556                 continue
557
558             cve['Relevant'] = True
559             filtered.append(cve)
560
561             if cve['Supersedes']:
562                 hotfixes.append(cve['Supersedes'])
563
564     # Collect patches that are already superseeded and
565     # merge these with the patches found installed on the system
566     hotfixes = ';'.join(set(hotfixes))
567
568     marked = set()
569     mark_superseeded_hotfix(filtered, hotfixes, marked)
570
571     # Check if left over KBs contain overlaps, for example a separate security hotfix
572     # which is also contained in a monthly rollup update
573     check = filter(lambda cve: cve['Relevant'], filtered)
574     supersedes = set([x['Supersedes'] for x in check])
575     checked = filter(lambda cve: cve['BulletinKB'] in supersedes, check)
576     for c in checked:
577         c['Relevant'] = False
578
579     # Final results
580     found = list(filter(lambda cve: cve['Relevant'], filtered))
581     for f in found:
582         del f['Relevant']
583
584     return filtered, found
585
586
587 # Function which recursively marks KBs as irrelevant whenever they are superseeded
588 def mark_superseeded_hotfix(filtered, superseeded, marked):
589     # Locate all CVEs for KB
590     for ssitem in superseeded.split(';'):
591         foundSuperseeded = filter(lambda cve: cve['Relevant'] and cve['BulletinKB'] ==
             ssitem, filtered)
592         for ss in foundSuperseeded:
593             ss['Relevant'] = False
594
595             # In case there is a child, recurse (depth first)
596             if ss['Supersedes'] and ss['Supersedes'] not in marked:
597                 marked.add(ss['Supersedes'])
598                 mark_superseeded_hotfix(filtered, ss['Supersedes'], marked)
599
600
601
602
```

```
603
604     # Determine Windows version based on the systeminfo input file provided
605     def determine_product(systeminfo):
606         systeminfo = charset_convert(systeminfo)
607
608         # Fixup for 7_sp1_x64_enterprise_fr_systeminfo_powershell.txt
609         systeminfo = systeminfo.replace('\xA0', '\x20')
610
611         # OS Version
612         regex_version = re.compile(r'.*?((\d+\.?){3}) ((Service Pack (\d)|N\/\w|.+) )?[
            -\xa5]+ (\d+).*',
613                                    re.MULTILINE | re.IGNORECASE)
614         systeminfo_matches = regex_version.findall(systeminfo)
615         if len(systeminfo_matches) == 0:
616             raise WesException(
617                 'Not able to detect OS version based on provided input file\n    In case
                    you used the missingpatches script, use: wes.py -m missing.txt')
618
619         systeminfo_matches = systeminfo_matches[0]
620         mybuild = int(systeminfo_matches[5])
621         servicepack = systeminfo_matches[4]
622
623         # OS Name
624         win_matches = re.findall('.*?Microsoft[\(R\)]{0,3} Windows[\(R\)?]{0,3}
            ?(Serverr? )?(\d+\.?\d?( R2)?|XP|VistaT).*',
625                                  systeminfo, re.MULTILINE | re.IGNORECASE)
626         if len(win_matches) == 0:
627             raise WesException('Not able to detect OS name based on provided input file')
628         win = win_matches[0][1]
629
630         # System Type
631         archs = re.findall('.*?([\w\d]+?)-based PC.*', systeminfo, re.MULTILINE | re.
            IGNORECASE)
632         if len(archs) > 0:
633             arch = archs[0]
634         else:
635             logging.warning('Cannot determine system\'s architecture. Assuming x64')
636             arch = 'x64'
637
638         # Hotfix(s)
639         hotfixes = get_hotfixes(systeminfo)
640
641         # Determine Windows 10 version based on build
642         version = None
643         for build in buildnumbers:
644             if mybuild == build:
645                 version = buildnumbers[build]
646                 break
647             if mybuild > build:
648                 version = buildnumbers[build]
649             else:
650                 break
651
652         # Compile name for product filter
653         # Architecture
654         if win not in ['XP', 'VistaT', '2003', '2003 R2']:
655             if arch == 'X86':
656                 arch = '32-bit'
657             elif arch == 'x64':
658                 arch = 'x64-based'
659
660
661
662
663
664
665
666
667
668
669
670
```

```
671
672         # Client OSs
673         if win == 'XP':
674             productfilter = 'Microsoft Windows XP'
675             if arch != 'X86':
676                 productfilter += ' Professional %s Edition' % arch
677             if servicepack:
678                 productfilter += ' Service Pack %s' % servicepack
679         elif win == 'VistaT':
680             productfilter = 'Windows Vista'
681             if arch != 'x86':
682                 productfilter += ' %s Edition' % arch
683             if servicepack:
684                 productfilter += ' Service Pack %s' % servicepack
685         elif win == '7':
686             productfilter = 'Windows %s for %s Systems' % (win, arch)
687             if servicepack:
688                 productfilter += ' Service Pack %s' % servicepack
689         elif win == '8':
690             productfilter = 'Windows %s for %s Systems' % (win, arch)
691         elif win == '8.1':
692             productfilter = 'Windows %s for %s Systems' % (win, arch)
693         elif win == '10':
694             productfilter = 'Windows %s Version %s for %s Systems' % (win, version, arch)
695         elif win == '11':
696             productfilter = 'Windows %s for %s Systems' % (win, arch)
697
698
699         # Server OSs
700         elif win == '2003':
701             if arch == 'X86':
702                 arch = ''
703             elif arch == 'x64':
704                 arch = ' x64 Edition'
705             pversion = '' if version is None else ' ' + version
706             productfilter = 'Microsoft Windows Server %s%s%s' % (win, arch, pversion)
707         # elif win == '2003 R2':
708         # Not possible to distinguish between Windows Server 2003 and Windows Server 2003
            R2 based on the systeminfo output
709         # See: https://serverfault.com/q/634149
710         # Even though in the definitions there is a distinction though between 2003 and
            2003 R2, there are only around 50
711         # KBs specificly for 2003 R2 (x86/x64) and almost 6000 KBs for 2003 (x86/x64)
712         elif win == '2008':
713             pversion = '' if version is None else ' ' + version
714             productfilter = 'Windows Server %s for %s Systems%s' % (win, arch, pversion)
715         elif win == '2008 R2':
716             pversion = '' if version is None else ' ' + version
717             productfilter = 'Windows Server %s for %s Systems%s' % (win, arch, pversion)
718         elif win == '2012':
719             productfilter = 'Windows Server %s' % win
720         elif win == '2012 R2':
721             productfilter = 'Windows Server %s' % win
722         elif win == '2016':
723             productfilter = 'Windows Server %s' % win
724         elif win == '2019':
725             productfilter = 'Windows Server %s' % win
726         elif win == '2022':
727             productfilter = 'Windows Server %s' % win
728         else:
729             raise WesException('Failed assessing Windows version {}'.format(win))
730
731         return productfilter, win, mybuild, version, arch, hotfixes
732
733
734
735
736
737
738
739
740
```

```
741
742     # Extract hotfixes from provided text file
743     def get_hotfixes(text):
744         hotfix_matches = re.findall('.*KB\d+.*', text, re.MULTILINE | re.IGNORECASE)
745         hotfixes = []
746         for match in hotfix_matches:
747             hotfixes.append(re.search('.*KB(\d+).*', match, re.MULTILINE | re.IGNORECASE).
                group(1))
748
749         return hotfixes
750
751
752     # Debugging feature to list hierarchy of superseeded KBs according to the definitions
    file
753     def debug_supersedes(cves, kbs, indent, verbose):
754         for kb in kbs:
755             # Determine KBs superseeded by provided KB
756             foundkbs = list(filter(lambda k: k['BulletinKB'] == kb, cves))
757
758             # Extract date and title
759             titles = []
760             for f in foundkbs:
761                 titles.append(f['Title'])
762             titles = list(set(filter(None, titles)))
763             titles.sort()
764
765             kbdate = foundkbs[0]['DatePosted'] if foundkbs else '????????'
766             kbtitle = titles[0] if titles else ''
767
768             # Print
769             indentstr = '  ' * indent
770             print('[%.2d][%s] %s%s - %s' % (indent, kbdate, indentstr, kb.ljust(7, ' '),
                kbtitle))
771             if verbose and len(titles) > 1:
772                 for t in titles[1:]:
773                     print('%s%s%s' % (indentstr, ' ' * 25, t))
774
775             # Recursively iterate over KBs superseeded by the current KB
776             supersedes = []
777             for f in foundkbs:
778                 supersedes += f['Supersedes'].split(';')
779             supersedes = list(set(filter(None, supersedes)))
780             debug_supersedes(cves, supersedes, indent + 1, verbose)
781
782
783     # Split up list of KBs and the potential Service Packs/Cumulative updates available
784     def get_patches_servicepacks(results, cves, productfilter):
785         # Extract available Service Packs (if any)
786         sp = list(filter(lambda c: c['CVE'].startswith('SP'), results))
787         if len(sp) > 0:
788             sp = sp[0]  # There should only be one result
789
790             # Only focus on OS + architecure, current service pack is not relevant
791             productfilter = re.sub(' Service Pack \d', '', productfilter)
792
793             # Determine service packs available for the OS and determine the latest
                version available
794             servicepacks = list(filter(lambda c: c['CVE'].startswith('SP') and
                productfilter in c['AffectedProduct'], cves))
795             lastpatch = get_last_patch(servicepacks, sp)
796
797             # Remove service packs from regular KB output
798             kbs = list(filter(lambda c: not c['CVE'].startswith('SP'), results))
799
800             return kbs, lastpatch
801
802         return results, None
803
804
805
806
807
```

```
808
809 def get_operatingsystems(found, os_name):
810     # Compile the list of operating systems available from the results of above filter
811     # This list is provided to the user to further filter down the specific
    vulnerabilities
812     allproducts = list(set(t['AffectedProduct'] for t in found))
813     regex_wp = re.compile('.*(Windows (Server|(\d+.?)+|XP).*)')
814     os_names = list(set([wp[0] for wp in regex_wp.findall('\n'.join(allproducts))]))
815     os_names.sort()
816
817     # If --os parameter is provided, filter results on OS
818     if os_name:
819         # Support for providing an index in stead of the full OS string
820         if os_name.isdigit():
821             if int(os_name) >= len(os_names):
822                 print(colored('[-] Invalid operating system index specified with the
                    --os parameter', 'red'))
823                 exit(1)
824             os_name = os_names[int(os_name)]
825
826     return os_names, os_name
827
828
829 def list_operatingsystems(os_names):
830     # List operating systems
831     print(colored('[I] List of operating systems:', 'green'))
832     i = 0
833     for name in os_names:
834         print('    [%d] %s' % (i, name))
835         i += 1
836
837
838 # Obtain most recent patch tracing back recursively locating records which
    superseeded the provided record
839 def get_last_patch(servicepacks, kb):
840     results = list(filter(lambda c: c['Supersedes'] == kb['BulletinKB'], servicepacks
    ))
841
842     if results:
843         return get_last_patch(servicepacks, results[0])
844     else:
845         return kb
846
847
848 # Show summary at the end of results containing the number of patches and the most
    recent patch installed
849 def print_summary(kbs, sp):
850     # Collect unique BulletinKBs
851     missingpatches = set(r['BulletinKB'] for r in kbs)
852     print(colored('[-] Missing patches: ', 'red') + '%s' % colored(len(missingpatches
    ), 'yellow'))
853
854     # Show missing KBs with number of vulnerabilites per KB
855     grouped = Counter([r['BulletinKB'] for r in kbs if r['DatePosted']])
856     foundmissing = grouped.most_common()
857     for line in foundmissing:
858         kb = line[0]
859         number = line[1]
860         print('    - KB%s: patches %s %s' % (kb, number, 'vulnerability' if number ==
            1 else 'vulnerabilities'))
861
862     # Show in case a service pack is missing
863     if sp:
864         print(colored('[-] Missing service pack', 'red'))
865         print('    - %s' % sp['Title'])
866
867
868
869
870
871
872
```

```
873
874         # Show additional missing KBs when the --missing parameter is used
875         if len(missingpatches) > len(grouped):
876             difference = missingpatches.symmetric_difference([r[0] for r in foundmissing])
877             for kb in difference:
878                 print('    - KB%s: patches an unknown number of vulnerabilities' % kb)
879             print(colored(
880                 '[I] Check the details of the unknown patches at
                    https://support.microsoft.com/help/KBID,\n    for example
                    https://support.microsoft.com/help/890830 in case of KB890830',
881                 'yellow'))
882
883         # Show date of most recent KB
884         # Skip if no most recent KB available
885         if len(grouped) == 0:
886             return
887         foundkb = get_most_recent_kb(kbs)
888         message = colored('[I] KB with the most recent release date', 'yellow')
889         print('%s\n'
890               '    - ID: KB%s\n'
891               '    - Release date: %s' % (message, foundkb['BulletinKB'], foundkb[
                  'DatePosted']))
892
893
894     # Obtain most recent KB from a dictionary of results
895     def get_most_recent_kb(results):
896         dates = [int(r['DatePosted']) for r in results if r['DatePosted']]
897         if dates:
898             date = str(max(dates))
899             return list(filter(lambda kb: kb['DatePosted'] == date, results))[0]
900         else:
901             return None
902
903
904     # Output results of wes.py to screen
905     def print_results(results):
906         print()
907         for res in results:
908             # Don't print KBs which are supplied through the --missing parameter but are
                not included in the definitions.zip
909             if not res['DatePosted']:
910                 continue
911
912             exploits = res['Exploits'] if 'Exploits' in res else ''
913             label = 'Exploit'
914             value = 'n/a'
915             if len(exploits) > 0:
916                 value = colored(exploits, 'blue')
917             if ',' in exploits:
918                 label = 'Exploits'
919
920             if res['Severity'] == 'Critical':
921                 highlight = 'red'
922             elif res['Severity'] == 'Important':
923                 highlight = 'yellow'
924             elif res['Severity'] == 'Low':
925                 highlight = 'green'
926             elif res['Severity'] == 'Moderate':
927                 highlight = 'blue'
928             else:
929                 highlight = 'red'
930
931
932
933
934
935
936
937
938
939
940
```

```
941
942             print('Date: %s\n'
943                   'CVE: %s\n'
944                   'KB: KB%s\n'
945                   'Title: %s\n'
946                   'Affected product: %s\n'
947                   'Affected component: %s\n'
948                   'Severity: %s\n'
949                   'Impact: %s\n'
950                   '%s: %s\n' % (res['DatePosted'], res['CVE'], res['BulletinKB'], res[
                      'Title'], res['AffectedProduct'],
951                                 res['AffectedComponent'], colored(res['Severity'],
                                    highlight), res['Impact'], label, value))
952
953
954     # Output results of wes.py to a .csv file
955     def store_results(outputfile, results):
956         print(colored('[+] Writing %d results to %s' % (len(results), outputfile), 'green'
            ))
957
958         # Python 2 compatibility
959         if sys.version_info.major == 2:
960             f = open(outputfile, 'wb')
961         else:
962             f = open(outputfile, 'w', newline='')
963
964         header = list(results[0].keys())
965         header.remove('Supersedes')
966         writer = csv.DictWriter(f, fieldnames=header, quoting=csv.QUOTE_ALL)
967         writer.writeheader()
968         for r in results:
969             if 'Supersedes' in r:
970                 del r['Supersedes']
971             writer.writerow(r)
972
973
974     # Validate file existence for user-provided arguments
975     def check_file_exists(value):
976         if not os.path.isfile(value):
977             raise argparse.ArgumentTypeError('File \'%s\' does not exist.' % value)
978
979         return value
980
981
982     # Validate file existence for definitions file
983     def check_definitions_exists(value):
984         if not os.path.isfile(value):
985             raise argparse.ArgumentTypeError(
986                 'Definitions file \'%s\' does not exist. Try running %s --update first.' %
                    (value, FILENAME))
987
988         return value
989
990     # Elhamdülillâh!
991
```