38
Like
Tweet
2
G+1

USING IDAPYTHON TO MAKE YOUR LIFE EASIER: PART 2

POSTED BY: Josh Grunzweig on December 30, 2015 12:00 PM

FILED IN: Unit 42
TAGGED: IDA Pro, IDAPython, malware

Continuing our theme of using IDAPython to make your life as a reverse engineer easier, I'm going to tackle a very common issue: shellcode and malware that uses a hashing algorithm to obfuscate loaded functions and libraries. This technique is widely used and analysts come across it often. Using IDAPython, we will take this challenging problem and defeat it quite easily.

BACKGROUND

Reverse engineers most often encounter obfuscated function names in shellcode. The process is quite simple overall. The code will initially load the kernel32.dll library at runtime. Then, it continues to use this loaded image to identify and store the LoadLibraryA function, which is used to load additional libraries and functions. This particular technique employs a hashing algorithm that is used to identify a function. The hashing algorithm is commonly CRC32, however, other variations, such as ROR13, are common as well.

While reverse engineering a piece of malware, I ran into the following technique:

```
.text:004125A0    push    1
.text:004125A2    push    7695D1CCh
.text:004125A7    push    edx
.text:004125A8    call    load_function
.text:004125AD    add     esp, 0Ch
.text:004125B0    mov     dword_41A59C, eax
.text:004125B5    cmp     eax, ebx
.text:004125B7    jz      loc_4124E6
.text:004125BD    push    1
.text:004125BF    push    0E62E824Dh
.text:004125C4    push    esi
.text:004125C5    call    load_function
.text:004125CA    add     esp, 0Ch
.text:004125CD    mov     dword_41A3D8, eax
.text:004125D2    cmp     eax, ebx
.text:004125D4    jz      loc_4124E6
.text:004125DA    push    1
.text:004125DC    push    9A80E589h
.text:004125E1    push    esi
.text:004125E2    call    load_function
.text:004125E7    add     esp, 0Ch
.text:004125EA    mov     dword_41A56C, eax
.text:004125EF    cmp     eax, ebx
.text:004125F1    jz      loc_4124E6
.text:004125F7    push    1
.text:004125F9    push    0F3B07FCCh
.text:004125FE    push    edi
.text:004125FF    call    load_function
.text:00412604    add     esp, 0Ch
.text:00412607    mov     dword_41A380, eax
.text:0041260C    cmp     eax, ebx
.text:0041260E    jz      loc_4124E6
.text:00412614    mov     edi, [ebp+var_18]
.text:00412617    push    1
.text:00412619    push    301BF0h
.text:0041261E    push    edi
.text:0041261F    call    load_function
.text:00412624    add     esp, 0Ch
.text:00412627    mov     dword_41A544, eax
.text:0041262C    cmp     eax, ebx
.text:0041262E    jz      loc_4124E6
.text:00412634    mov     eax, [ebp+var_4]
.text:00412637    push    1
.text:00412639    push    0A9290135h
.text:0041263E    push    eax
.text:0041263F    call    load_function
.text:00412644    add     esp, 0Ch
.text:00412647    mov     dword_41A38C, eax
```

Figure 1 Malware loading functions dynamically using CRC32 hash

In the above example, we were able to quickly identify the constant of 0xEDB88320, which is used by the CRC32 algorithm.

Home
Government
Partners
Unit 42 Threat Intelligence
Technical Documentation
Advanced Endpoint Protection

unit42

Get Updates
Sign up to receive the latest news, cyber threat intelligence and research from Unit 42.

Business Email

Submit

SUBSCRIBE TO THE RESEARCH CENTER BLOG

Subscribe

CATEGORIES & ARCHIVES

Select a Category

Select a Month

MORE →

RECENT POSTS

Using IDAPython to Make Your Life Easier: Part 4
posted by Josh Grunzweig on January 6, 2016

Customer Spotlight Video: Better Control and Visibility at Animal Logic
posted by Rae Harrison on January 5, 2016

Palo Alto Networks Researcher Discovers Critical Vulnerabilities in Adobe Flash
posted by Ryan Olson on January 5, 2016

The Power of App-ID
posted by Stephen Perciballi on January 4, 2016

Using IDAPython to Make Your Life Easier: Part 3
posted by Josh Grunzweig on January 4, 2016

MORE →

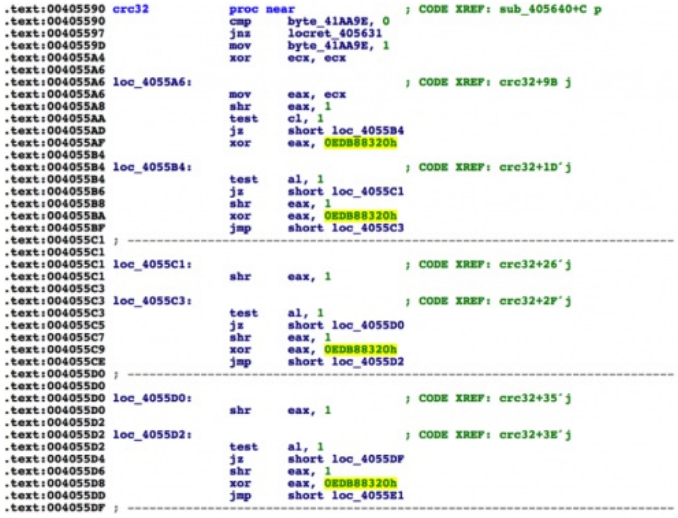

Figure 2 CRC32 algorithm identified

Now that the algorithm and function have been identified, we can look at the number of cross-references to determine how many times this function is called. In total, this function is called 190 times in this particular sample. Clearly, decoding all of these hashes and renaming them by hand within our IDA Pro file is not something we wish to perform. As such, we can use IDAPython to make our lives easier.

SCRIPTING IN IDAPYTHON

The first step actually does not use IDAPython whatsoever, but it does use Python. In order to identify what hashes equate to what functions, we need to generate a list of the most common function hashes on a Microsoft Windows operating system. To do this, we can simply grab a list of common libraries used by the Windows OS, and iterate over their function names.

```
1  def get_functions(dll_path):
2    pe = pefile.PE(dll_path)
3    if ((not hasattr(pe, 'DIRECTORY_ENTRY_EXPORT')) or (pe.DIRECTORY_
4      print "[*] No exports for %s" % dll_path
5      return []
6    else:
7      expname = []
8      for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
9        if exp.name:
10         expname.append(exp.name)
11     return expname
```

We can then take this list of function names and perform the CRC32 hashing algorithm against it.

```
1 def calc_crc32(string):
2   return int(binascii.crc32(string) & 0xFFFFFFFF)
```

Finally, we write the results to a JSON-formatted file, which I've named 'output.json'. This JSON data file contains a large dictionary, using the following format:

```
1 HASH => NAME
```

A full copy of this script can be foundhere.

Once this file has been generated, we can go back to IDA, where the remainder of our scripting will take place. The first step of our script will be to read the JSON data from the previously created 'output.json'. Unfortunately, JSON objects to not support using integers as the key value, so after this data is loaded, we modify the keys to represent integers instead of strings.

```
1 for k,v in json_data.iteritems():
2   json_data[int(k)] = json_data.pop(k)
```

After this data has been properly loaded, we're going to create a new enumeration that will store the hash-to-function-name mapping. (For more information about enumerations and how they work, I encourage you to read this tutorial.)

Using enumerations, we're able to map an integer value, such as a CRC32 hash, to a string representation, such as a function name. In order to create a new enumeration in IDA, we make use of the AddEnum() function. To make the script more versatile, we first check to see if the enumeration already exists, using the GetEnum() function.

```
1 enumeration = GetEnum("crc32_functions")
2 if enumeration == 0xFFFFFFFF:
3   enumeration = AddEnum(0, "crc32_functions", idaapi.hexflag())
```

This enumeration will be modified later on. The next step will be to determine what cross-references our function responsible for converting hashing to function has. This should look

familiar to those that have read part 1. When looking at the structure of how functions are passed to the function, we see that the CRC32 hash is provided as the second argument.

```
.text:004124F3                push    1
.text:004124F5                push    4C8A5B22h
.text:004124FA                push    edx
.text:004124FB                call    load_function
```

Figure 3 Arguments being passed to load_function

As such, we're going to iterate through the previous instructions leading up to the function call, looking for the second instance of a push instruction. Once discovered, we check the CRC32 hash against our previously loaded JSON data from output.json to ensure this value has a function name associated with it.

```
for x in XrefsTo(load_function_address, flags=0):
    current_address = x.frm
    addr_minus_20 = current_address-20
    push_count = 0
    while current_address >= addr_minus_20:
      current_address = PrevHead(current_address)
      if GetMnem(current_address) == "push":
        push_count += 1
        data = GetOperandValue(current_address, 0)
        if push_count == 2:
          if data in json_data:
            name = json_data[data]
```

At this point, we add this CRC32 hash and function name to our previously created enumeration, using the AddConstEx() function.

```
AddConstEx(enumeration, str(name), int(data), -1)
```

Once this data has been added to the enumeration, we can convert our CRC32 hash to its enumeration name. The following two functions can be used to acquire the first instance of a number to its enumeration, as well as convert data at a certain address to this enumeration.

```
def get_enum(constant):
  all_enums = GetEnumQty()
  for i in range(0, all_enums):
    enum_id = GetnEnum(i)
    enum_constant = GetFirstConst(enum_id, -1)
    name = GetConstName(GetConstEx(enum_id, enum_constant, 0, -1))
    if int(enum_constant) == constant: return [name, enum_id]
    while True:
      enum_constant = GetNextConst(enum_id, enum_constant, -1)
      name = GetConstName(GetConstEx(enum_id, enum_constant, 0, -1))
      if enum_constant == 0xFFFFFFFF:
        break
      if int(enum_constant) == constant: return [name, enum_id]
  return None

def convert_offset_to_enum(addr):
  constant = GetOperandValue(addr, 0)
  enum_data = get_enum(constant)
  if enum_data:
    name, enum_id = enum_data
    OpEnumEx(addr, 0, enum_id, 0)
    return True
  else:
    return False
```

After this enumeration conversion has taken place, we're going to take a look at renaming the DWORD that holds the address of the discovered function after it is loaded.

```
.text:004124F3                push    1
.text:004124F5                push    4C8A5B22h
.text:004124FA                push    edx
.text:004124FB                call    load_function
.text:00412500                add     esp, 0Ch
.text:00412503                mov     dword_41A36C, eax
```

Figure 4 Storing function address to DWORD after it is loaded

To do this, we will iterate not before the function, but after, looking for an instruction that is moving eax to a DWORD. When this is discovered, we'll rename this DWORD to the correct function name. To avoid naming conflicts, we will prepend 'd_' to the name.

```
address = current_address
while address <= address_plus_30:
  address = NextHead(address)
  if GetMnem(address) == "mov":
    if 'dword' in GetOpnd(address, 0) and 'eax' in GetOpnd(address,
      operand_value = GetOperandValue(address, 0)
      MakeName(operand_value, str("d_"+name))
```

Putting this all together, we're able to update the previous unreadable disassembly to something much more readable.

Figure 5 Changes after running script

Now, when we look at the list of DWORDs being used to store this information, we get a list of actual function names. This data can then be used to perform additional static analysis against the malware in question.

```
.data:0041A510 d_func_advapi32_cryptgethashparam dd ?  ; DATA XREF: sub_40ADB0+68↑r
.data:0041A510                                         ; sub_4122E0+12E4↑w
.data:0041A514 d_func_kernel32_resumethread dd ?       ; DATA XREF: sub_4122E0+928↑w
.data:0041A518 d_func_shlwapi_urlescapea dd ?          ; DATA XREF: sub_4121E0+78↑r
.data:0041A518                                         ; sub_4122E0+1532↑w
.data:0041A51C d_func_wininet_internetconnecta dd ?    ; DATA XREF: sub_40B7F0+73↑r
.data:0041A51C                                         ; sub_4122E0+CC1↑w
.data:0041A520 d_func_user32_flashwindow dd ?          ; DATA XREF: sub_404DB0+75↑r
.data:0041A520                                         ; sub_4122E0+7F1↑w
.data:0041A524 d_func_kernel32_getlasterror dd ?       ; DATA XREF: sub_4030E0+1FB↑r
.data:0041A524                                         ; sub_403F40+683↑r ...
.data:0041A528 d_func_msvcrt_malloc dd ?               ; DATA XREF: .text:00406A89↑r
.data:0041A528                                         ; sub_406BE0+1B7↑r ...
.data:0041A52C d_func_kernel32_getmodulefilenamea dd ? ; DATA XREF: sub_4030E0+62↑r
.data:0041A52C                                         ; sub_4122E0+8CE↑w ...
.data:0041A530 d_func_kernel32_thread32first dd ?      ; DATA XREF: sub_4122E0+CA1↑w
.data:0041A534 d_func_ntdll_memcmp dd ?                ; DATA XREF: sub_403F40+58B↑r
.data:0041A534                                         ; sub_409890+130↑r ...
.data:0041A538 d_func_ntdll_ntquerysysteminformation dd ? ; DATA XREF: sub_404710+33↑r
.data:0041A538                                         ; sub_4122E0+FEE↑w
.data:0041A53C d_func_ntdll_memcpy dd ?                ; DATA XREF: sub_402A90+A5↑r
.data:0041A53C                                         ; sub_402A90+167↑r ...
.data:0041A540 d_func_ntdll_strncat dd ?               ; DATA XREF: sub_40BD80+9C↑r
.data:0041A540                                         ; sub_4122E0+512↑w
.data:0041A544 d_func_ntdll_zwqueueapcthread dd ?      ; DATA XREF: sub_40B220+CE↑r
.data:0041A544                                         ; sub_4122E0+347↑w
.data:0041A548 d_func_kernel32_process32first dd ?     ; DATA XREF: sub_40C160+42↑r
.data:0041A548                                         ; sub_4122E0+649↑w
.data:0041A54C d_func_kernel32_getcurrentprocessid dd ? ; DATA XREF: sub_402EF0+61↑r
.data:0041A54C                                         ; sub_404A20:loc_404AFB↑r ...
.data:0041A550 d_func_advapi32_regenumvaluew dd ?      ; DATA XREF: sub_4122E0+6C3↑w
.data:0041A554 d_func_ntdll_rtlgetversion dd ?         ; DATA XREF: sub_404A20+6D↑r
.data:0041A554                                         ; sub_4122E0+1643↑w
.data:0041A558 d_func_kernel32_createtoolhelp32snapshot dd ? ; DATA XREF: sub_40C160+E↑r
.data:0041A558                                         ; sub_4122E0+73D↑w
.data:0041A55C d_func_ole32_coinitializesecurity dd ?  ; DATA XREF: sub_405DD0+36↑r
.data:0041A55C                                         ; sub_4122E0+293↑w
.data:0041A560 d_func_oleaut32_sysallocstring dd ?     ; DATA XREF: sub_405EE0+7E↑r
```

Figure 6 DWORD naming after script runs

The full IDAPython script can be found here.

CONCLUSION

We were once again able to take a fairly difficult task of being provided with 190 CRC32 hash representations of function names and extracting meaningful data using IDAPython. Enumerations can be a powerful mechanism when faced with such a problem. Creating and modifying enumerations, and assigning enumerations to variables can be easily performed using IDAPython, saving us valuable time. Additionally, these enumerations can be exported and imported into other IDA projects in the event an analyst finds the same challenge while reverse engineering another sample.

***Addendum 12/31/2015*:** As pointed out to me by Alex Hanel on Twitter, in the event you rename the function to its actual function name, IDA Pro will automatically perform parameter propagation. This adds an additional level of information to the analyst, making static analysis easier. A simple modification to the script mentioned earlier in this blog post will allow the analyst to perform this action. In the event the name already exists, simply add a '_' or '_1' to it in order to avoid conflicts.

POST YOUR COMMENT

Name *

Email *

Website

Post Comment

Copyright © 2007-2013 Palo Alto Networks