VB2021 localhost

7 - 8 October, 2021 / vblocalhost.com

BUGS IN MALWARE – UNCOVERING VULNERABILITIES FOUND IN MALWARE PAYLOADS

Nirmal Singh & Uday Pratap Singh

Zscaler, India

nsingh@zscaler.com
upratap@zscaler.com

ABSTRACT

Malware authors often take advantage of vulnerabilities in popular software and use various techniques to bypass security products like anti-virus, sandbox, and intrusion detection systems. Security researchers find ways to patch such bugs in products to make effective detection statically and dynamically. There has been a lot of research on anti-VM and anti-sandbox techniques and techniques for bypassing AV products, but we haven't seen much on the opposite side: finding bugs in pieces of malware that stop them from spreading and infecting the system. Just like legitimate applications, malware is also prone to bugs and coding errors which can cause it to crash or which can serve as backdoors for whitehats to undo the damage. Such bugs can often persist in a family for a long time.

In this research we look at multiple prevalent malware families in which we uncovered various coding errors. The purpose of this research is threefold:

1. To look at what type of vulnerabilities exist in some of the prevalent malware families.
2. To discuss the use of these bugs/vulnerabilities in preventing malware infection.
3. To find out whether these are real vulnerabilities/coding errors or escape mechanisms.

INTRODUCTION

We observed that sometimes malware doesn't validate the output of a queried API or is unable to handle different types of C&C response. Authors often develop malware according to their local environment and don't take into consideration techniques that may be present in target environments, such as ASLR and DEP, causing the malware to crash.

To illustrate multiple bugs and coding errors in malware, we have performed a large-scale analysis on a data set of malicious samples collected from the *Zscaler Cloud Sandbox* based on a few behaviour signatures. We collected such samples from late 2019 to March 2021 in the *Zscaler Cloud*. The files were clustered based on the behaviour of malware observed in *Zscaler Cloud Sandbox* and given names accordingly.

The graph in Figure 1 shows data from the *Cloud Sandbox* over a six-month period. Of 500K+ samples marked as malware by *Cloud Sandbox* during this period, 8,800+ samples showed execution errors.

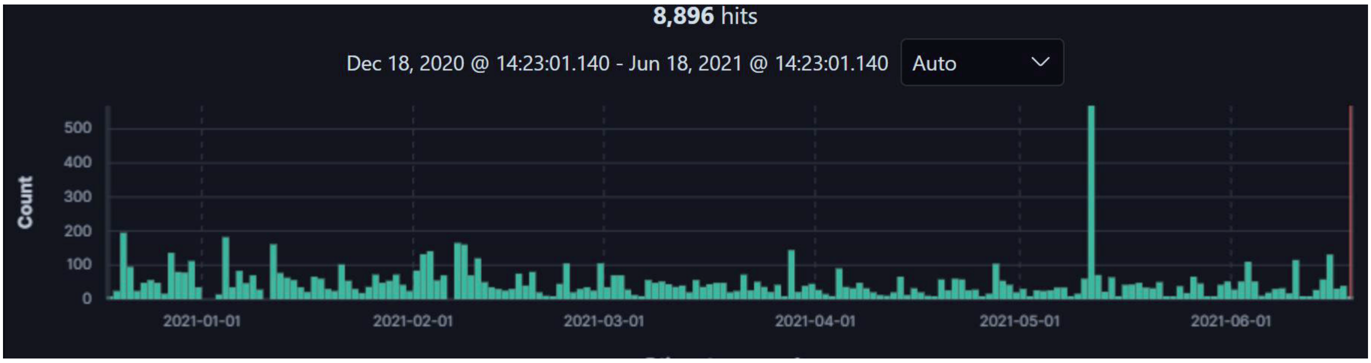

Figure 1: Malware showing execution errors.

In our research we found several malware families with a common set of bugs in their code and we found that sometimes a single malware family has multiple bugs, providing a number of opportunities for security researchers to help victims.

We found that not all, but a few bugs can be helpful in preventing or cleaning infection, stopping encryption and the spreading of malware if they are used as a kill-switch in a local system. We will cover the details of the kill-switch in a few cases where a user can create certain files with certain privileges or add an additional registry entry into the system.

Malware authors are constantly upgrading their code and making it hard to analyse and detect using sandboxes and other security products. Sometimes such changes and enhancements lead to coding errors.

In this paper we will cover different types of bugs in malware samples and divide those into a number of categories based on MITRE's Common Weakness Enumeration (CWE) system [1]:

- CWE-131: Incorrect Calculation of Buffer Size
- CWE-253: Incorrect Check of Function Return Value
- CWE-787: Out-of-bounds Write
- CWE-253: Incorrect Check of Function Return Value
- CWE-390: Detection of Error Condition Without Action
- CWE-622: Improper Validation of Function Hook Arguments

- CWE-444: Inconsistent Interpretation of HTTP Requests
- CWE-280: Improper Handling of Insufficient Permissions or Privileges
- CWE-913: Improper Control of Dynamically-Managed Code Resources
- CWE-1023: Incomplete Comparison with Missing Factors
- CWE-474: Use of Function with Inconsistent Implementations

CASE STUDY 1: WIN32.PWS.VIDAR – MULTIPLE BUGS IN THE CODE

Vidar, also known as Vidar stealer, is a dangerous piece of malware that steals information and cryptocurrency from infected users. It derives its name from the ancient Scandinavian god of vengeance. Besides credit card numbers and passwords, Vidar can also scrape an impressive selection of digital wallets. In the *Zscaler Cloud Sandbox*, we found 94 samples showing execution errors (Figure 2).

Figure 2: Number of Vidar samples showing execution errors.

During our analysis of Vidar we found three bugs which caused the malware to crash. Details of the bugs are given below:

Bug 1: Incorrect check of function return value

This bug is about calling an API and performing an operation without validating the output of that API call. The registry key shown below is related to *WinSCP* software, Vidar steals stored credentials in this registry key:

`HKEY_CURRENT_USER\Software\Martin Prikryl\WinSCP 2\Sessions\Default%20Settings`

Vidar uses the RegGetValueA API to extract a password from the registry path, but it doesn't verify whether the call was successful, as can be seen in Figure 3.

```
push    offset aPassword_1 ; "Password"
lea     eax, [ebp+0D78h+Name]
push    eax
push    [ebp+0D78h+phkResult] ; HKEY_CURRENT_USER\Software\
                              ; Martin Prikryl\WinSCP 2\
                              ; Sessions\Default%20Settings
mov     [ebp+0D78h+var_D7C], ebx
call    esi ; byte_473020 ; RegGetValueA
mov     ecx, [ebp+0D78h+var_D98] ; Not return code check
lea     eax, [ebp+0D78h+var_D08]
push    eax             ; void *
lea     eax, [ebp+0D78h+var_908]
push    eax             ; int
lea     eax, [ebp+0D78h+var_508]
push    eax             ; int
lea     eax, [ebp+0D78h+var_D5C]
push    eax             ; int
call    DecryptPassWord
```

Figure 3: Password extraction from registry key.

It further tries to decrypt the password and makes a call to the strcpy_s vc++ runtime function with invalid parameters, which results in the process crashing. This can be used as a kill-switch by keeping the above registry entry empty and stopping infection for Vidar samples.

This bug is part of CWE-253 and it has consequences such as unexpected state, DoS, crash, exit, or restart of the system.

Bug 2: Common buffer used by an API to perform multiple tasks & out-of-bounds write

We found another bug in Vidar where an API used the same buffer with restricted size to download and read the payload. In one of the samples, spotted in February 2021, it downloads config files from the C&C using the InternetReadFile *Windows*

API. As shown in the code snapshot in Figure 4, InternetReadFile uses the same buffer for downloading the subsequent data. So it will corrupt the data downloaded earlier if the data size is more than 2,047 bytes (this size is defined in the code). In this case the malware will not be able to download the correct config file.

```
IReadFile_Loop:                         ; CODE XREF: DownLoadConfig+FF↓j
                mov     eax, [ebp+804h+dwNumberOfBytesRead]
                cmp     eax, ebx
                jz      short loc_404FBE ; Exit Loop if dwNumberOfBytesRead is zero
                mov     [ebp+eax+804h+Buffer], bl
                lea     eax, [ebp+804h+dwNumberOfBytesRead]
                push    eax             ; lpdwNumberOfBytesRead
                push    edi             ; dwNumberOfBytesToRead = 0x000007FF
                lea     eax, [ebp+804h+Buffer]
                push    eax             ; lpBuffer
                push    [ebp+804h+hFile] ; hFile

loc_404FB8:                             ; CODE XREF: DownLoadConfig+E2↑j
                call    esi ; InternetReadFile
                test    eax, eax
                jnz     short IReadFile_Loop
```

Figure 4: C2 communication.

This bug is a classic case of CWE-787 where malware writes data past the end of the buffer, which results in the corruption of data, a crash, or code execution.

Bug 3: Detection of absent string in configuration without any action

The malware sample has another bug that crashes it if it's not able to download data from the C&C or if it's not able to find a specific string ('about') in the downloaded data. In the code snapshot shown in Figure 5, the function *FindStrLocation* finds the location of a string stored in the *field* variable. The code inside the *if* statement executes if the string is found. The *crashHere* function is outside of the *if* statement but uses the return value of the *strtok* function. The *strToken* variable will be NULL if *FindStrLocation* is not able to find the string and returns *-1*. This will crash the sample.

```
v2 = DownLoadConfig((int)&v6, *(LPCSTR *)&v7, v8, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19);
LOBYTE(v20) = 3;
sub_401704(&v14, (void *)v2);
sub_4013B4(&v6, 1, 0);
LOBYTE(v20) = 0;
sub_4013B4(&v7, 1, 0);
v3 = FindStrLocation((int)&v14, (const char *)field, 0);
if ( v3 != -1 )
{
  sub_40133E(0, v3 + 8);
  v4 = v14;
  if ( v19 < 0x10 )
    v4 = (char *)&v14;
  strToken = strtok(v4, v5);
}
crashHere(&dword_486078, strToken);
sub_4013B4(&v14, 1, 0);
```

Figure 5: C2 response parsing.

Here, we refer to CWE-390, where the malware detects an error but doesn't perform any action to prevent the consequences of the error, which may result in sample crashing.

CASE STUDY 2: WIN32.DOWNLOADER.RUGMI – INCORRECT CALCULATION OF BUFFER SIZE

Rugmi is a downloader which has been seen downloading RATs, e.g. Remcos, and other malware. We saw 17 samples of this malware showing execution errors during a campaign that was active from February to March 2021. This malware usually downloads a PNG file from i[.]imgur[.]com, which contains configuration data and a payload file. The data inside the PNG file is compressed and encrypted. The decryption logic assumes that the size of the uncompressed data will be four times the size of the file, so it allocates memory according to that (see Figure 6).

```
push    esi
call    eax             ; GetFileSize
mov     ebx, eax
call    GETDLL
push    esi
mov     edx, 0B09315F4h
mov     ecx, eax
call    GETAPI
call    eax             ; CloseHandle
test    ebx, ebx
jz      short loc_49E76C1
lea     esi, ds:0[ebx*4] ; FileSize*4
test    esi, esi
jz      short loc_49E76C1
call    GETDLL
mov     edx, 9CE0D4Ah
mov     ecx, eax
call    GETAPI
push    4
push    3000h
push    esi
push    0
call    eax             ; VirtualAlloc
mov     esi, eax
```

Figure 6: Calculation of buffer size.

The malware allocates a buffer four times its size, but sometimes the size of the decrypted file is bigger than that. In such cases, the malware crashes when trying to extract the embedded data due to buffer overflow (see Figure 7).

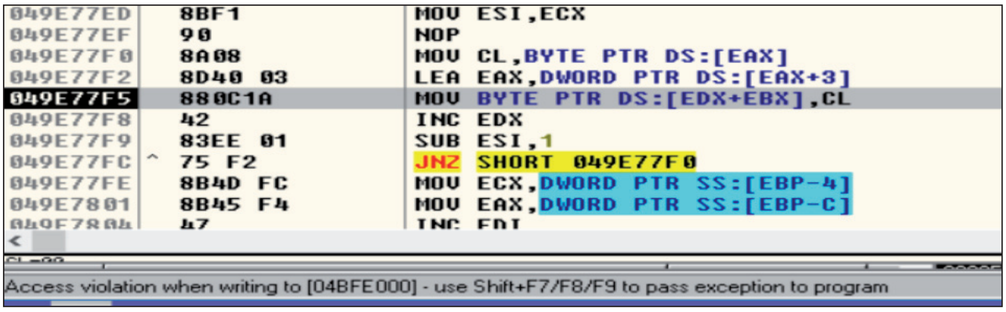

Figure 7: Access violation during decryption.

We map this bug with CWE-131. Such bugs may lead to an out-of-bounds read or write, possibly causing a crash, allowing arbitrary code execution, or exposing sensitive data.

CASE STUDY 3: WIN32.TROJAN.BUERLOADER – LOADING UNVALIDATED RESOURCE LOCATION TABLE

Buerloader is a first-stage malware, active from mid-2019 and seen in the wild downloading other ransomware and banking malware. In June 2020, we came across an interesting variant of Buerloader which was crashing during its execution. We found 19 samples of this variant showing similar behaviour and all were leading to crashes due to similar bugs.

For installation, this sample drops itself in the %PROGRAMDATA% folder and starts a new instance with following command-line parameters:

```
C:\ProgramData\Ostersin\gennt.exe "<initial file location>" ensgJJ
```

It starts the secinit.exe legitimate process in suspended mode using the CreateProcessW API. It allocates new memory in the target process and writes DLL and initialization code for DLL using the VirtualAlloc and WriteProcessMemory APIs, respectively. Finally, it starts a remote thread using the RtlCreateUserThread API.

The DLL initialization code performs the following actions:

1. Fixes the DLL offset using the relocation table in the PE header.
2. Parses the import table of the DLL and loads the DLLs mentioned in the import table using the LdrLoadDll *Windows* API.

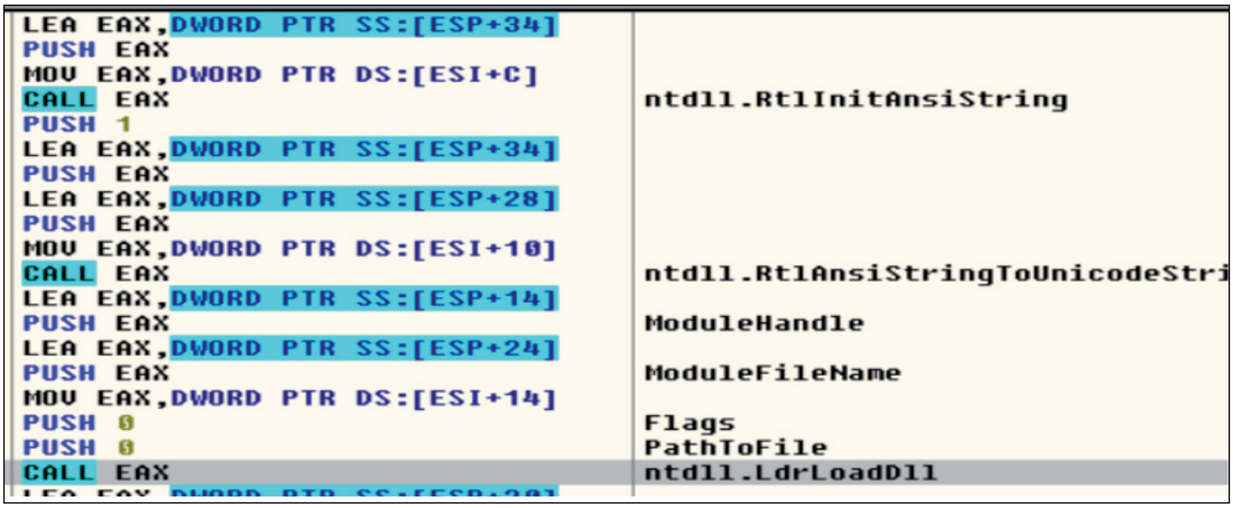

Figure 8: DLL initialization code.

3. Builds the import table using the LdrGetProcedureAddress API.
4. Calls the entry point of the DLL.

The issue here is that the DLL file is compiled with IMAGE_FILE_RELOCS_STRIPPED, meaning it can't be loaded on any random address, so it crashes on loading.

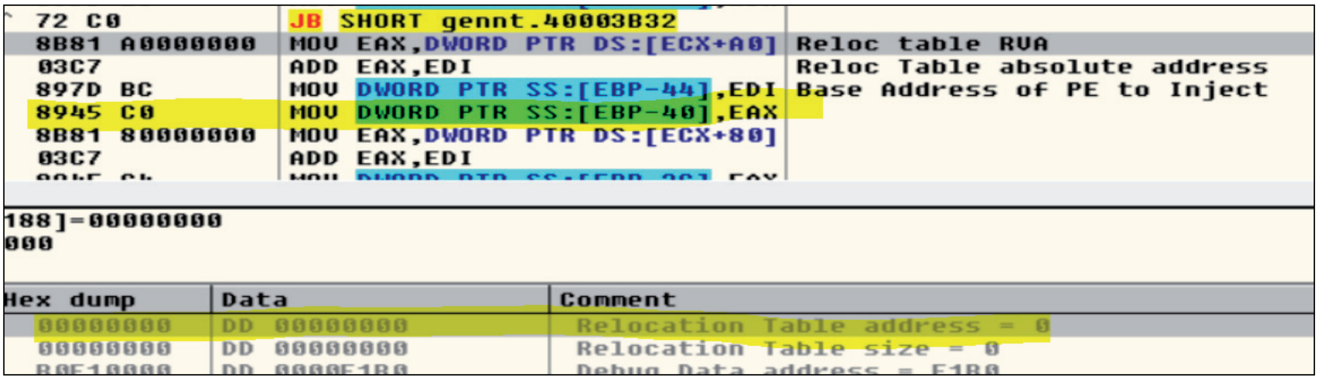

Figure 9: Relocation table parsing.

The code shown in Figure 9 indicates that the injector doesn't check whether the relocation table address is present in the PE header. This results in incorrect relocation calculations when the DLL initialization code loads the DLL in the target process.

According to *MSDN* [2], if relocation information was stripped from the file, then the file must be loaded at its preferred base address. If the base address is not available, the loader reports an error. This bug falls under CWE-913, which relates to improper control of dynamically managed code resources, in this case the relocation table.

CASE STUDY 4: WIN32.PWS.OSKI – INCORRECT CHECK OF FUNCTION RETURN VALUE

Oski, introduced in 2019, is a piece of malware with the capability of stealing personal and sensitive information from a victim's system [3]. The name 'Oski' is derived from an old Nordic word meaning Viking warrior, which is quite fitting considering this popular info-stealer is extremely effective at pillaging privileged information from its victims.

It also steals passwords stored in *Google Chrome*. It copies the 'Login Data' file from the location '%LOCALAPPDATA%\Google\Chrome\User Data\Default' in 'C:\ProgramData\<InstallFolder>\tmp'. The Login Data file is a *SQLite* database and the malware extracts the following information from the login table: origin_url, username_value and password_value (see Figure 10).

This malware uses SQLITE3.DLL APIs for extraction of the information from the login table. It uses the following APIs: sqlite3_open, sqlite3_prepare_v2, sqlite3_step, sqlite3_column_text and sqlite3_column_bytes. It uses the sqlite3_column_text API to extract the data from the first two columns. For the 'password_value' column it first checks the available data length using sqlite3_column_bytes (see Figure 11).

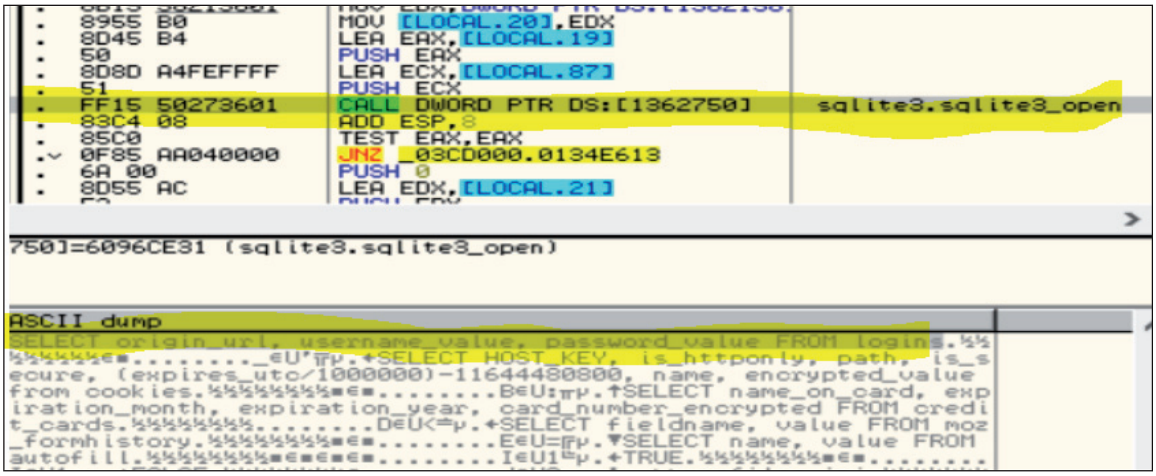

Figure 10: SQL query for extracting passwords.

```
push    2                   ; column ID = 2
mov     eax, [ebp+psqlite3_stmt]
push    eax
call    sqlite3_column_bytes
add     esp, 8
push    eax                 ; Size
push    2
mov     ecx, [ebp+psqlite3_stmt]
push    ecx
call    sqlite3_column_blob
add     esp, 8
push    eax                 ; Src
lea     edx, [ebp+decryptBuffer]
push    edx                 ; int
call    DecryptData
add     esp, 14h
```

Figure 11: Code to extract and decrypt password.

As is clear from the code snapshot in Figure 11, it does not verify the return length from the sqlite3_column_bytes API. Now, if the 'password_value' column is NULL, the sqlite3_column_bytes API will also return Null, which will result in the sqlite3_column_blob API returning a NULL pointer. The same NULL pointer will be used in the DecryptData function, causing the application to crash due to the null data reference exception. So we can create a kill-switch for this malware by inserting dummy data and keeping the 'password_value' as NULL in the 'Login Data' file using tools like *DB Browser for SQLite* [4] or using the APIs from SQLITE3.DLL. We found a couple of Oski samples that had similar bugs.

This bug is covered under CWE-253, in which an unexpected return value may result in a crash or the exit of the malware. Normally, a stealer expects a username & password pair to exist in the browser database. But a user can create an entry with an empty password, which will act as a kill-switch which may stop a stealer with this bug.

CASE STUDY 5: MULTIPLE MALWARE FAMILIES – INCONSISTENT INTERPRETATION OF HTTP RESPONSE HANDLING FOR RELATED APIs

We encountered multiple samples where the HTTP response from threat actor-controlled systems is not validated and received data is used directly by the malware, leading to a crash. A number of bugs in different APIs are described in the following sections.

Bug 1: Win32.Downloader.Penguish – no check for InternetReadFile API output

This is a downloader sample and it shows an execution error when it encounters an unexpected HTTP response from the C2 server as it doesn't validate the C2 response read through the InternetReadFile *Windows* API. We found 100+ similar samples from this family.

The *C2_Communication* function (see Figure 12) uses InternetReadFile for downloading C2 data. This function returns a value of 1 if it is able to download the C2 data and 0 otherwise. The downloaded data and size of data is stored in the *ptr_structInternetData.ptr_C2DataFull* and *ptr_structInternetData.dwSizeOfc2Data* pointers, respectively.

```
  if ( !ptr_C2Data )
  {
    BuildMachineID(sz);
    v6 = xmmword_34D800;
    v7 = 35;
    v3 = Decrypt_C2_path(&v6);
    wsprintfA(byte_388800, (LPCSTR)v3, sz);
    *(_QWORD *)&v6 = 8532196438026516344i64;
    DWORD2(v6) = 2053405564;
    WORD6(v6) = 31356;
    BYTE14(v6) = 0;
    ptr_structInternetData.ptr_unKwn2 = sub_361AB5(&v6);
    ptr_structInternetData.c2portNumber = 8055;
    ptr_structInternetData.ptr_ToC2Path = (int)byte_388800;
    ptr_structInternetData.flag1 = 1;
    C2_Communication(&ptr_structInternetData);
    v2 = sz;
    ptr_C2Data = (char *)ptr_structInternetData.ptr_C2DataFull;
    *(_BYTE *)(ptr_structInternetData.dwSizeOfc2Data + ptr_structInternetData.ptr_C2DataFull) = 0;
  }
  BuildMachineID(v2);
  memmove_0(&CopyOF_structInternetData, &ptr_structInternetData, 0x20u);
  CopyOF_structInternetData.ptr_PingStr = (int)v1;
  CopyOF_structInternetData.pingStrLen = strlen((const char *)v1) + 1;
  sub_3689C7();
  C2_Communication(&CopyOF_structInternetData);
  sub_3689C7();
  return CopyOF_structInternetData.ptr_C2DataFull;
}
```

Figure 12: C2 communication.

Figure 12 shows the pseudocode of the function. As you will see, there is no check for a return value from the *C2_Communication* function and it tries to modify the data stored in *ptr_C2DataFull* (as highlighted in Figure 12), but if there is no data available then it will crash due to a null reference exception.

Bug 2: Win32.Downloader.Glupteba – no check for URLDownloadToFile API output

This is a downloader which downloads the well-known malware Glupteba. This sample calls the URLDownloadToFile API to download samples and the *C2_Talk* function to download C2 data. A code snapshot is shown in Figure 13.

```
                lea     ecx, [ebp+C2_Data]
;   } // starts at 403D08
;   try {
                mov     byte ptr [ebp+var_4], 15h
                call    C2_Talk
                lea     edx, [ebp+C2_Data]
;   } // starts at 403D25
;   try {
                mov     byte ptr [ebp+var_4], 17h
                lea     ecx, [ebp+DecryptedData]
                call    DecryptData
                add     esp, 18h
;   } // starts at 403D34
;   try {
                mov     byte ptr [ebp+var_4], 18h
                mov     ecx, esp        ; this
                lea     eax, [ebp+DecryptedData]
                push    eax             ; Src
                and     dword ptr [ecx+10h], 0
                and     dword ptr [ecx+14h], 0
                call    std__string__copy_ctor
```

```
push    edi
mov     edi, edx
mov     esi, ecx
xor     ecx, ecx
lea     eax, [ebp+arg_0]
push    ecx
mov     [ebp+var_4], ecx
cmp     [ebp+arg_14], 10h
push    ecx
cmovnb  eax, [ebp+arg_0]
push    esi
push    eax
push    ecx
call    ptr_URLDownloadToFileA
mov     ebx, ds:Sleep
push    3E8h            ; dwMilliseconds
call    ebx ; Sleep
push    ecx
mov     edx, edi
mov     ecx, esi        ; lpFileName
call    sub_8633D7
pop     ecx
push    64h ; 'd'       ; dwMilliseconds
call    ebx ; Sleep
```

Figure 13: Downloading payload.

However, it doesn't check for return values for the *URLDownloadToFile* API and the *C2_Talk* function. If the malware payload is not available or the C2 response does not result in the desired output, it crashes the sample.

Both above bugs described here are related to the misinterpretation of HTTP response, which falls under CWE-444. These pieces of malware expect a specific response from the C2 server, but cannot handle invalid responses.

CASE STUDY 6: WIN32.BACKDOOR.EMOTET – WILDCARD SEARCH OF DLL WITH A SINGLE CHARACTER

Emotet, a famous malware-as-a-service (MaaS), was first seen in 2014. It was mainly spread through malspam and is known as a strain of banking trojan. It was taken down by law enforcement agencies in January 2021. In *Zscaler Cloud*

Sandbox, we found a few Emotet samples which had a very critical issue in the logic they used to get the address of system DLLs. We found 318 Emotet samples showing execution errors due to different types of bugs, as explained below.

One of the samples, spotted in August 2020, has an issue in the logic it uses to get the address of the NTDLL.DLL system DLL. It uses Process Environment Block (PEB) to get the image base of the required DLL and then uses a custom GetProcAddress-like function to retrieve the address of an exported function from the DLL.

```
int __cdecl GetModHandle(unsigned __int16 *dllName)
{
  int PEB_offset; // ST10_4@1
  int InLoadOrderModuleListBase; // [sp+0h] [bp-10h]@1
  int InLoadOrderModuleListCurrent; // [sp+Ch] [bp-4h]@1

  PEB_offset = *(_DWORD *)(__readfsdword(0x30u) + 0xC);
  InLoadOrderModuleListBase = *(_DWORD *)(PEB_offset + 0xC);
  InLoadOrderModuleListCurrent = *(_DWORD *)(PEB_offset + 0xC);
  do
  {
    if ( !CompareBaseDLLName(*(unsigned __int16 **)(InLoadOrderModuleListCurrent + 0x30), dllName) )
      return *(_DWORD *)(InLoadOrderModuleListCurrent + 0x18);// Return DLL Base Address
    InLoadOrderModuleListCurrent = *(_DWORD *)InLoadOrderModuleListCurrent;
  }
  while ( InLoadOrderModuleListCurrent != InLoadOrderModuleListBase );
  return 0;
}
```

Figure 14: Code to get module handle of NTDLL.DLL.

As can be seen in the pseudocode in Figure 14, it uses *InLoadOrderModuleList* to get the base name of the module and compares only the module name, not the extension. Now, if the current process name is ntdll.exe, it will be at the top of the list in *InLoadOrderModuleList.*

The other interesting thing in the Emotet installation logic is that it chooses a file name randomly from the files in the %SYSTEM32% folder and copies itself with that name to the SYSWOW64 folder. It uses the *SHGetFolderPathW* API with CLSID as CSIDL_SYSTEMX86 to get the full path. This directory also contains the NTDLL.DLL file, so the ntdll.exe process name is possible, and in that case it will crash because the malware payload doesn't have an export table and the custom GetProcAddress function doesn't verify whether the module contains an export table and tries to read unavailable memory area.

A similar issue was found in another sample but for a different DLL. Here, if we change the file name to anything that starts with 'K' it will result in the crash. Here also, the issue lies in the logic that extracts the image base of KERNEL32.DLL.

```
PEB_offset = *(_DWORD *)(__readfsdword(0x30u) + 0xC);
InLoadOrderModuleListBase = *(_DWORD *)(PEB_offset + 0xC);
InLoadOrderModuleListCurrent = *(_DWORD *)(PEB_offset + 0xC);
while ( 1 )
{
  v4 = hashKey;

  DLL_Base_Name = _wcslwr(*(wchar_t **)(InLoadOrderModuleListCurrent + 0x30));// DLL_Base_Name

  char_dllName = *(_BYTE *)DLL_Base_Name;
  i = v4
  while(char_dllName)
  {

    i = char_dllName + 0x32 * i; //Calculate hash                1

    char_dllName = *((_BYTE *)DLL_Base_Name + 1);

    DLL_Base_Name = (wchar_t *)((char *)DLL_Base_Name + 1) //Next char      2

  }
  if ( i == dllNameHash )                                         3
    break;
  InLoadOrderModuleListCurrent = *(_DWORD *)InLoadOrderModuleListCurrent;
  if ( InLoadOrderModuleListCurrent == InLoadOrderModuleListBase )
    return 0;
}
return *(_DWORD *)(InLoadOrderModuleListCurrent + 0x18); //Return Image Base
```

Figure 15: Code to get module handle of KERNEL32.DLL.

Similar to the earlier issue, it uses *InLoadOrderModuleList* to get the base DLL name. The base name is converted to lowercase and a hash is calculated (point 1 in Figure 15) using all the characters in the DLL name. The DLL name is a wide-character string, so to get the next character, you have to add two bytes to the base pointer. However, in the code only one is added (point 2 in Figure 15). This results in the loop exiting just after the first character (point 3 in Figure 15). And it will return the image base even if only one character matches. This results in a crash if the process name starts with ‘K’ because it also uses a custom GetProcAddress similar to the one mentioned above.

During our more in-depth research, we found a similar issue in another sample that uses the same logic for extracting the image base of NTDLL.DLL, so in this case if the sample name starts with ‘N’, it will crash the sample.

In all three samples, we found an incomplete comparison for different DLL names. Such bugs are covered under CWE-1023 and may lead to altered execution logic, bypass of protection mechanism, etc.

CASE STUDY 7: WIN32.PWS.RACCOON – USE OF FUNCTION WITH INCONSISTENT IMPLEMENTATIONS

Malware samples are usually packed using unknown packers. This helps the malware to avoid detection. Sometimes, however, it can also result in the failure of the malware installation or impact other functionality. We saw as example of this in a variant of the Raccoon malware. Raccoon stealer [5] is a type of malware focused on gathering sensitive information from the infected system. This malware is also known to steal financial and user-specific information.

One of the information-stealing capabilities of Raccoon is to extract and steal credentials stored by *Internet Explorer*. Starting with *Windows 7*, *Internet Explorer* stores sensitive information including passwords in the *Windows Vault* [6]. To extract the passwords from the Vault, this malware uses different APIs (VaultOpenVault, VaultCloseVault, VaultEnumerateItems, VaultGetItem and VaultFree) from VAULTCLI.DLL.

There is a change in the VaultGetItem API starting from *Windows 8*, so you have to check the version of the OS before using the correct version of the API. This malware uses the GetVersionExW API to get the OS version details (see Figure 16). It checks for the major version to be 6 and the minor version to be 2 or greater.

```
mov     [ebp+VersionInformation.dwOSVersionInfoSize], esi
lea     eax, [ebp+VersionInformation]
push    eax             ; lpVersionInformation
call    ds:GetVersionExW
cmp     [ebp+VersionInformation.dwMajorVersion], 6
jnz     short loc_427811
cmp     [ebp+VersionInformation.dwMinorVersion], 2
mov     [ebp+bWin8OrGreater], 1
jnb     short loc_427814

                        ; CODE XREF: GotoCrash+55↑j
mov     [ebp+bWin8OrGreater], bl
```

Figure 16: Code to get Windows version information.

However, as per *MSDN* documentation, the behaviour of this API has changed, starting from *Windows 8.1*. For applications not manifested for *8.1* or *Windows 10*, this API will always return the *Windows 8* OS version value (6.2). Since the original Raccoon payload is not manifested for *8.1* or *Windows 10*, it works fine on *Windows 10*.

The packer used in the sample that we analysed was actually manifested for *Windows 7*, *8*, *8.1* and *10* (see Figure 17).

```
<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
<application>
<!-- Windows 10 -->
<supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/>
 <!-- Windows 8.1 -->
<supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>
<!-- Windows 8 -->
<supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/>
 <!-- Windows 7 -->
<supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>
</application>
</compatibility>
</assembly>
```

Figure 17: Packer manifest file.

This packer injects the Raccoon payload into another instance of itself, so the GetVersionExW API will provide an accurate version of the *Windows* OS, which will be 10 for major version and 0 for the minor version for *Windows 10*. The Raccoon code is designed to check only the 6.2 version, and this will result in selecting the wrong version of the VaultGetItem API and a crash. We found 1,000+ samples crashing due to this bug.

We consider this bug under CWE-474, which states that the code uses a function that has inconsistent implementations across operating systems and versions. The common consequences of this bug may be high likelihood that a weakness will be exploited to achieve a certain impact, but a low likelihood that it will be exploited to achieve a different impact.

CASE STUDY 8: WIN32.RANSOM.SAPPHIRE – IMPROPER HANDLING OF INSUFFICIENT PERMISSIONS OR PRIVILEGES

Ransomware is a type of malware that encrypts a victim's files and demands a ransom from the victim to restore access to the data upon payment. The Sapphire ransomware encrypts files with the .VIVELAG extension. It encrypts all files in the 'C:\' directory and skips files with the .VIVELAG extension. We encountered a couple of samples that were showing an execution error.

We found a variant of this ransomware that doesn't check the permission of directories before adding them to a list which is later used to encrypt the files, and it causes a crash.

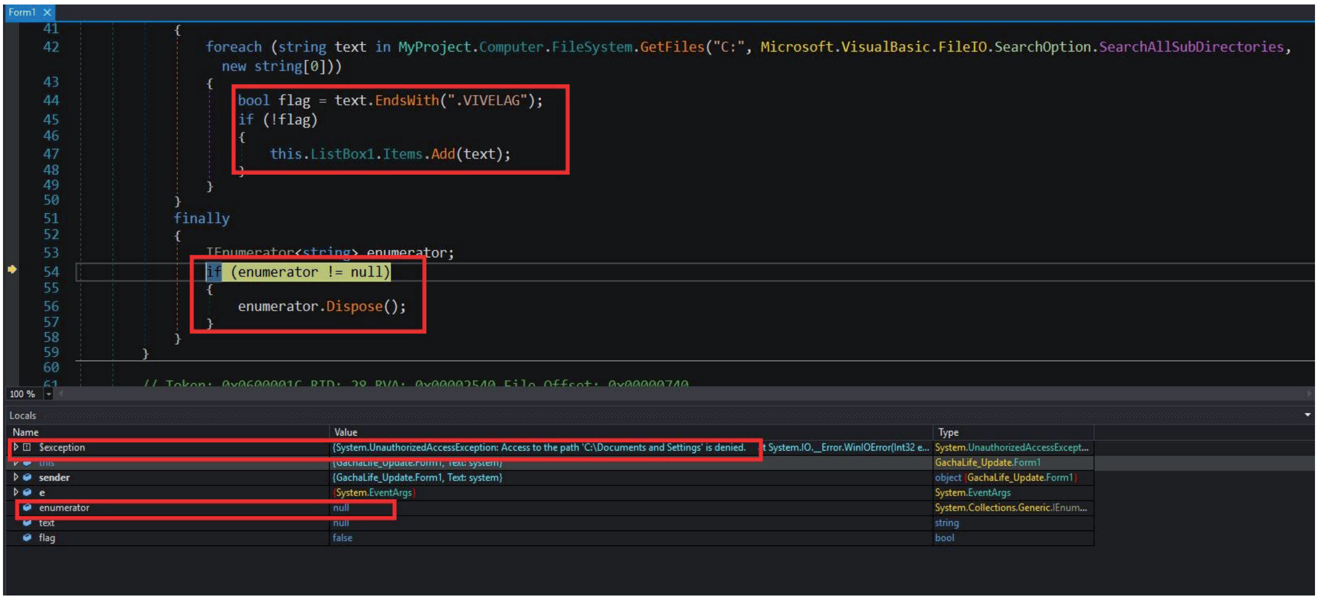

Figure 18: Code to enumerate files in C drive.

As the code in Figure 18 shows, the malware enumerates all files and directories under the 'C:\' directory and adds the file path in ListBox1. But during enumeration, the malware doesn't check the privileges of the directory 'Documents and Settings' and doesn't handle the exception. This ransomware sample expects a list of files and folders to encrypt but it ends up encrypting nothing. So, we can say that due to this bug, it is a variant of ransomware that doesn't encrypt anything but shows warnings of encryption and demands a ransom.

Figure 19: Unhandled exception.

This bug can be used as a kill-switch for such ransomware by creating a directory under all drives with protected permissions. If the ransomware doesn't perform checks for permissions, it will end up doing nothing. We can see the exception window and ransom banner in Figure 19. This ransomware disables task managers, so the victim can use a third-party tool to kill the process showing the ransomware banner.

We class this bug under CWE-280, in which a program doesn't handle, or handles incorrectly, insufficient privileges to access resources or functionality as specified by their permissions. This may cause it to follow unexpected code paths that may leave the application in an invalid state.

CASE STUDY 9: MISCELLANEOUS

In this section we present different case studies which have different types of bugs, e.g. improper check of downloaded data, improper check of exported function by a DLL, and try to perform different types of operation on that data.

1: Malware name: Win32.Trojan.Agent

This sample is a component of another piece of malware and is used to load a dropped DLL and execute the exported function *rtrrtrtrtrt*. But as is clear from the code shown in Figure 20, it doesn't verify the return values from APIs and crashes if the DLL is not present or the export function is not found.

```
push    ebp
mov     ebp, esp
push    ecx
push    offset ProcName ; "rtrrtrtrtrt"
mov     eax, 4
shl     eax, 0
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+eax]
push    edx             ; lpLibFileName
call    ds:LoadLibraryW
push    eax             ; hModule
call    ds:GetProcAddress
mov     [ebp+var_4], eax ; No return value check
mov     eax, 4
shl     eax, 1
mov     ecx, [ebp+arg_4]
mov     edx, [ecx+eax]
push    edx
call    [ebp+var_4]
```

Figure 20: Calling export function of DLL.

2: Malware name: Win32.Downloader.RemcosRAT

This is a downloader sample, it downloads encrypted payloads from cdn[.]discordappp[.]com and drive[.]google[.]com. The downloader is compiled in Delphi and has a malicious DLL file embedded that it loads in memory. This DLL file actually downloads the encrypted payload, decrypts it, and loads it. It uses simple XOR-based decryption logic (see Figure 21).

```
mov     eax, [ebp+DecryptionKey]
call    sub_694698
test    eax, eax
jle     short loc_6A356C
mov     eax, [ebp+DecryptionKey]
call    sub_694698
push    eax
mov     eax, ds:dword_6A68FC
pop     edx
mov     ecx, edx
cdq
idiv    ecx
inc     edx
mov     eax, [ebp+DecryptionKey]
mov     al, [eax+edx-1]
xor     al, [edi]
mov     [edi], al
```

Figure 21: Decryption logic.

The decrypted payload should be a PE file. It extracts the information of important fields (like Image size) from the PE header and allocates memory according to that.

However, we found it crashing in the sandbox. The reason for the crash is that it does not check whether the decrypted payload is a PE file or not. It just tries to parse the junk data as a PE file and crashes (Figure 22).

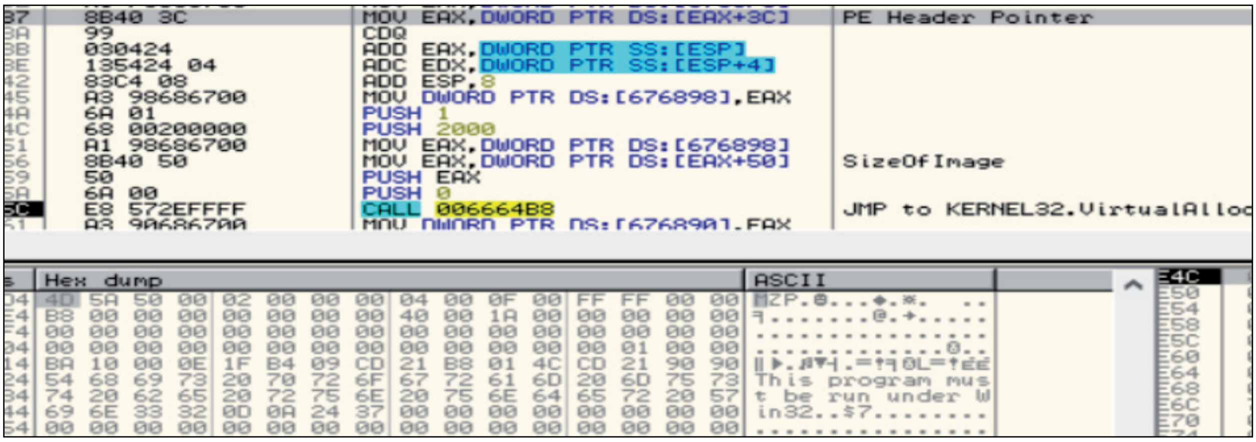

Figure 22: PE loader.

3: Win32.Backdoor.RemcosRAT – invalid memory access to load resource

RemcosRAT emerged in 2016 and gives threat actors complete control over the target system. It can steal data, keys and digital wallets and can run surveillance, e.g. audio or screenshots.

We found this sample in February 2021 and it was showing execution errors. Upon analysis, we found a bug in the code where it doesn't verify the return value from different API calls. Basically, it loads the configuration data from the resource section of the executable but doesn't check whether the resource is present (Figure 23).

```
push    ebp
mov     ebp, esp
push    esi
push    edi
push    0Ah               ; lpType
push    offset aSettings  ; "SETTINGS"
push    0                 ; hModule
call    ds:FindResourceA  ; No return Code check
mov     edi, eax
push    edi               ; hResInfo
push    0                 ; hModule
call    ds:LoadResource   ; No return Code check
push    eax               ; hResData
call    ds:LockResource   ; No return Code check
push    edi               ; hResInfo
push    0                 ; hModule
mov     esi, eax
call    ds:SizeofResource
mov     ecx, [ebp+arg_0]
pop     edi
mov     [ecx], esi
```

Figure 23: Code to load configuration data from resource section.

After loading the data it tries to decrypt the malware configuration and access memory location which leads to the crash (Figure 24).

```
call    load_resource
mov     [ebp+Size], eax
mov     eax, [ebp+var_8]
movzx   ebx, byte ptr [eax] ; Crash here
push    ebx             ; Size
call    ds:malloc
```

Figure 24: Memory allocation.

This bug is also covered under CWE-253, in which a function return value is not validated properly, causing the malware sample to crash.

CONCLUSION

In this research, we looked at multiple examples of malware with different types of vulnerabilities which cause crashes and also provide opportunity for a user to use them as a kill-switch. We tried to classify all the bugs using MITRE's CWE list, which makes it easy to get a more detailed definition and consequences of bugs. This study includes a broad range of malware from stealers and downloaders to ransomware. This research shows that malware code often contains multiple bugs and indicates that no proper quality assurance checks are performed on malware code. Security vendors can leverage these bugs to write different types of signatures to identify and block such malware attacks.

REFERENCES

[1] CWE Common Weakness Enumeration. CWE VIEW: Software Development. https://cwe.mitre.org/data/definitions/699.html.

[2] Microsoft. IMAGE_FILE_HEADER structure (winnt.h). May 2018. https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_file_header.

[3] Cohen, B. Meet Oski Stealer: An In-depth Analysis of the Popular Credential Stealer. Cyberark. https://www.cyberark.com/resources/threat-research-blog/meet-oski-stealer-an-in-depth-analysis-of-the-popular-credential-stealer.

[4] DB Browser for SQLite. https://sqlitebrowser.org/.

[5] Analyzing the Raccoon stealer. Cyberark. https://lp.cyberark.com/rs/316-CZP-275/images/CyberArk-Labs-Racoon-Malware-wp.pdf.

[6] Haephrati, M. The Secrets of Internet Explorer Credentials. Code Project. January 2017. https://www.codeproject.com/Articles/1167943/The-Secrets-of-Internet-Explorer-Credentials.