Parasite HTTP RAT cooks up a stew of stealthy tricks

proofpoint.com/us/threat-insight/post/parasite-http-rat-cooks-stew-stealthy-tricks

July 25, 2018

Parasite HTTP RAT cooks up a stew of stealthy tricks

July 25, 2018 Proofpoint Staff

Overview

Proofpoint researchers recently discovered a new remote access Trojan (RAT) available for sale on underground markets. The RAT, dubbed Parasite HTTP, is especially notable for the extensive array of techniques it incorporates for sandbox detection, anti-debugging, anti-emulation, and other protections. The malware is also modular in nature, allowing actors to add new capabilities as they become available or download additional modules post infection.

To date, we have only observed Parasite HTTP in a single small email campaign with intended recipients primarily in the information technology, healthcare, and retail industries.

Campaign Analysis

On July 16, 2018, Proofpoint observed a small campaign that appeared to leverage human resources distribution lists as well as some individual recipients at a range of organizations. Specifically, the campaign targeted the following distribution lists, among others:

- hr@[organization domain]
- recruiting@
- accessibility@
- resumes@

The messages purported to be resumes or CV submissions and used subjects including:

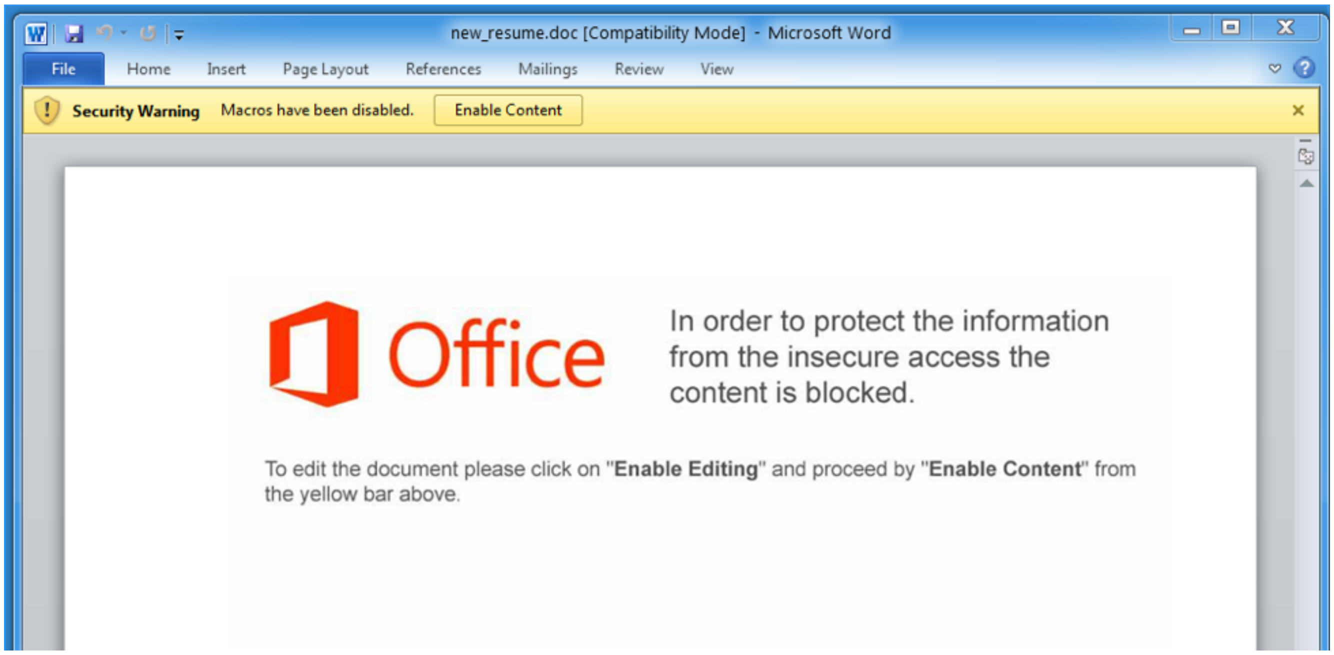
- advertised position
- would like to apply
- application

The messages contained Microsoft Word attachments with names such as (Figure 1):

- my_cv.doc
- resume_.doc
- cvnew.doc
- cv.doc

- new_resume.doc

The documents contained macros that, if enabled, would download Parasite HTTP from a remote site.



In order to protect the information from the insecure access the content is blocked.

To edit the document please click on "**Enable Editing**" and proceed by "**Enable Content**" from the yellow bar above.

Figure 1: Document attachment

Parasite HTTP Forum Advertisement

The first insight into the new malware observed in the campaign above came from an advertisement on an underground forum. The Parasite HTTP ad details a long list of sophisticated features, many of which focus on evading detection and analysis. The full text of the advertisement appears below:

****

Parasite HTTP Remote Administration Tool

What is Parasite HTTP?

Parasite HTTP is a professionally coded modular remote administration tool for windows written in C that has no dependencies except the OS itself.

With the stub size of ~49kb and plugin support it presents perfect solution for controlling large amount of computers from a remote location.

Features

- No dependencies (Coded in C)
- Small stub size (~49kb uncompressed, ~23kb compressed)
- Dynamic API calls (No IAT)
- Encrypted strings
- Bypass Ring3 hooks
- Secure C&C panel written in PHP
- Firewall bypass

- Supports both x86 and x64 Windows OS (from XP to 10)
- Full unicode support
- Online builder tied to your domain/s (Build bot bin anytime with any settings you wish)
- Encrypted communication with C&C panel (Optional - SSL using self signed certificate)
- Plugin system
- Multiple backup domains
- System wide persistence (x86 processes only) (Optional)
- Injection to whitelisted system process (Optional)
- Install & Melt (Optional)
- Hidden startup (Optional)
- Anti-Emulation (Optional)
- Anti-Debug
- Extended statistics and informations in the panel
- Advanced task management system
- On Connect task (New clients will execute task/s)
- Low resource usage
- Special login page security code
- Captcha on login page to prevent brute force attacks
- Download & Execute (Supports both HTTP and HTTPS links)
- Update
- Uninstall

Available Plugins (Some are created by me)

- User management
- Browser password recovery
- FTP password recovery
- IM password recovery
- Email password recovery
- Windows licence keys recovery
- Hidden VNC
- Reverse Socks5 proxy

System Req

- PHP 5.6 or greater (Gd & OpenSSL)
- IonCube Loader
- SQL Database

****

Malware Analysis

Parasite HTTP contains an impressive collection of obfuscation and sandbox- and research environment-evasion techniques. In this section we examine the more sophisticated components.

String Obfuscation

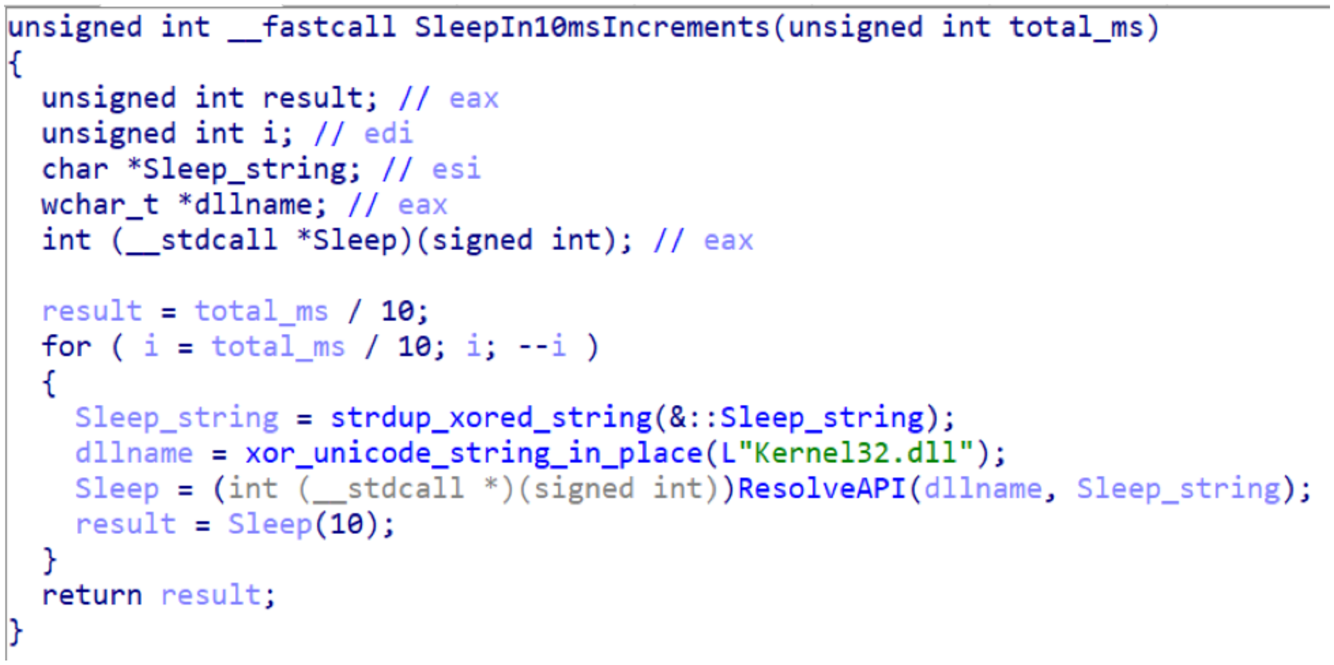
Parasite HTTP uses the following encoding for its obfuscated strings:

- 4-byte XOR key
- 2-byte string length in characters
- N-byte obfuscated ASCII/Unicode string

This 6-byte header exists prior to the reference to the string in assembly. Parasite HTTP contains four routines for operating on strings preceded by a 6-byte header. For each type of string, ASCII or Unicode, one variant leaves the obfuscated string in place and returns a dynamically-allocated, deobfuscated version of the string. The other variant uses VirtualProtect to deobfuscate the string in place, setting the XOR key to 0 after the deobfuscation has been performed, which effectively skips deobfuscation during future access to the string.

Sandbox Detection via Sleep Manipulation

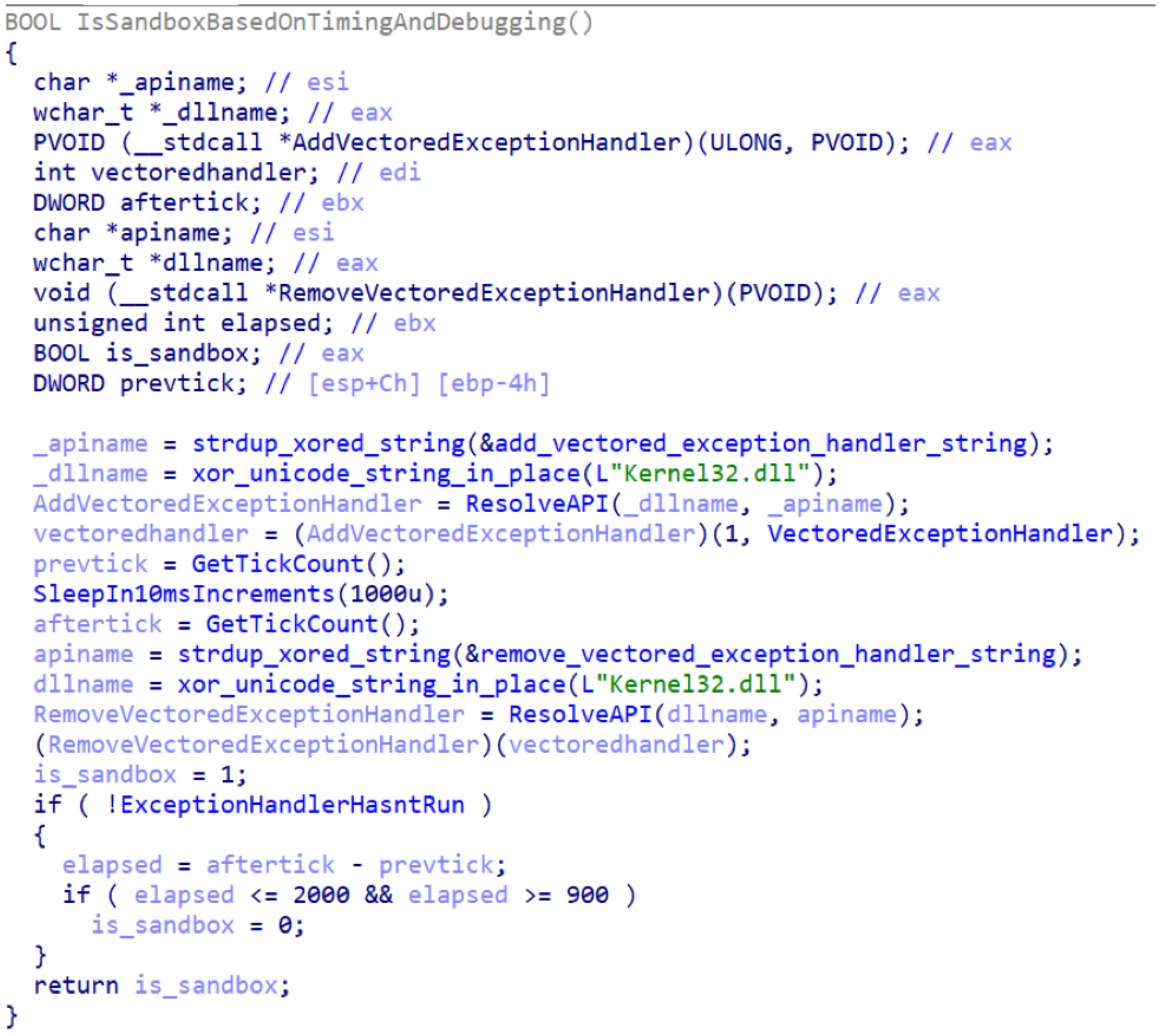
Parasite HTTP uses a sleep routine to delay execution and check for sandboxes or emulation. Specifically it employs the following code to perform its sleeps:



```
unsigned int __fastcall SleepIn10msIncrements(unsigned int total_ms)
{
  unsigned int result; // eax
  unsigned int i; // edi
  char *Sleep_string; // esi
  wchar_t *dllname; // eax
  int (__stdcall *Sleep)(signed int); // eax

  result = total_ms / 10;
  for ( i = total_ms / 10; i; --i )
  {
    Sleep_string = strdup_xored_string(&::Sleep_string);
    dllname = xor_unicode_string_in_place(L"Kernel32.dll");
    Sleep = (int (__stdcall *)(signed int))ResolveAPI(dllname, Sleep_string);
    result = Sleep(10);
  }
  return result;
}
```

Figure 2: Routine that Parasite HTTP uses to perform sleeps in 10ms increments (decompiled and with functions renamed for readability)

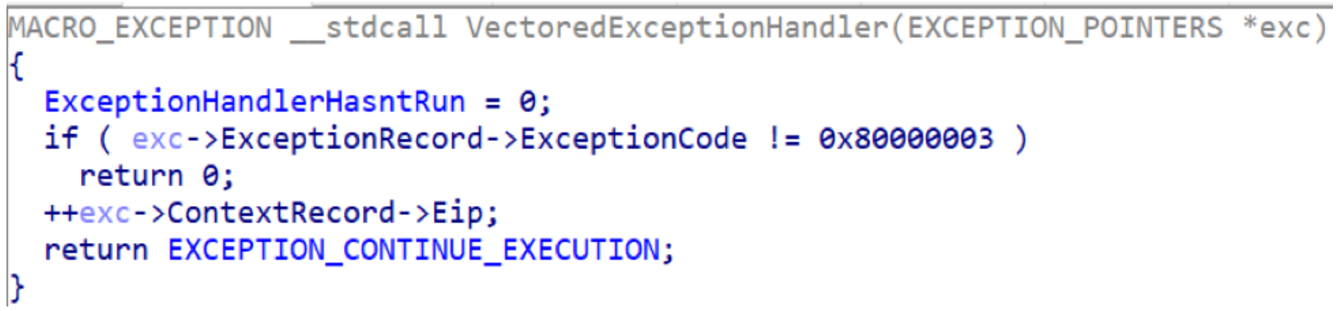
The above function is called by a routine that performs a sandbox check for the expected passage of time as well as non-interference with its own handling of breakpoint instructions it has placed in the checking routine (removed from the display below for readability):



```
BOOL IsSandboxBasedOnTimingAndDebugging()
{
  char *_apiname; // esi
  wchar_t *_dllname; // eax
  PVOID (__stdcall *AddVectoredExceptionHandler)(ULONG, PVOID); // eax
  int vectoredhandler; // edi
  DWORD aftertick; // ebx
  char *apiname; // esi
  wchar_t *dllname; // eax
  void (__stdcall *RemoveVectoredExceptionHandler)(PVOID); // eax
  unsigned int elapsed; // ebx
  BOOL is_sandbox; // eax
  DWORD prevtick; // [esp+Ch] [ebp-4h]

  _apiname = strdup_xored_string(&add_vectored_exception_handler_string);
  _dllname = xor_unicode_string_in_place(L"Kernel32.dll");
  AddVectoredExceptionHandler = ResolveAPI(_dllname, _apiname);
  vectoredhandler = (AddVectoredExceptionHandler)(1, VectoredExceptionHandler);
  prevtick = GetTickCount();
  SleepIn10msIncrements(1000u);
  aftertick = GetTickCount();
  apiname = strdup_xored_string(&remove_vectored_exception_handler_string);
  dllname = xor_unicode_string_in_place(L"Kernel32.dll");
  RemoveVectoredExceptionHandler = ResolveAPI(dllname, apiname);
  (RemoveVectoredExceptionHandler)(vectoredhandler);
  is_sandbox = 1;
  if ( !ExceptionHandlerHasntRun )
  {
    elapsed = aftertick - prevtick;
    if ( elapsed <= 2000 && elapsed >= 900 )
      is_sandbox = 0;
  }
  return is_sandbox;
}
```

Figure 3: Function that detects sandbox environments by checking for the passage of time and non-interference with its own handling of breakpoint instructions

During the interval between the installation and removal of the vectored exception handler, several breakpoint instructions are executed in the function which triggers execution of this vectored exception handler, shown below. That handler sets the global flag which notifies the parent function that the handler has run and skips over the breakpoint instruction. This code has been copied verbatim from an example of anti-debugging code posted in a public repository [1].



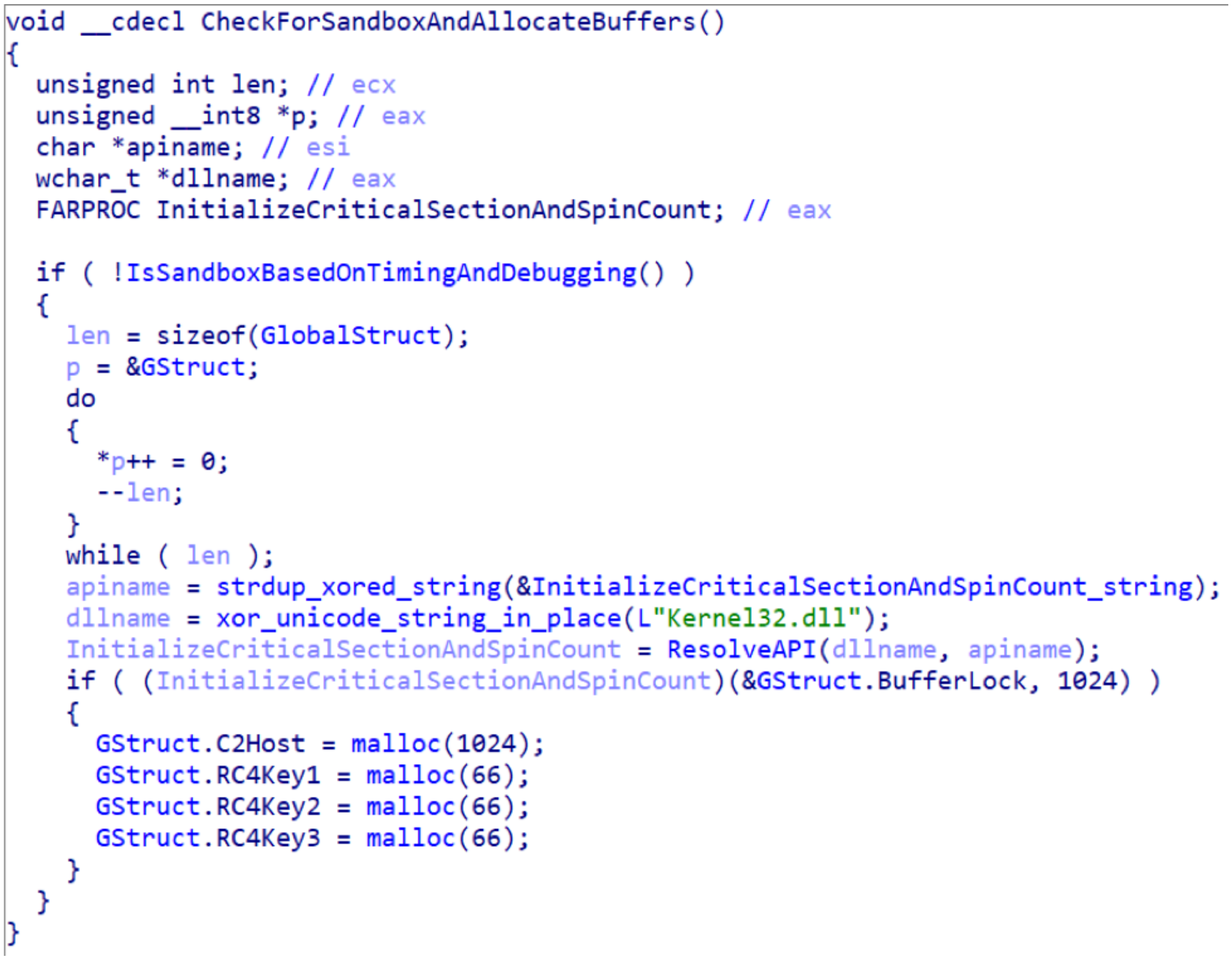
```
MACRO_EXCEPTION __stdcall VectoredExceptionHandler(EXCEPTION_POINTERS *exc)
{
  ExceptionHandlerHasntRun = 0;
  if ( exc->ExceptionRecord->ExceptionCode != 0x80000003 )
    return 0;
  ++exc->ContextRecord->Eip;
  return EXCEPTION_CONTINUE_EXECUTION;
}
```

Figure 4: Exception handler

The sandbox checking routine first checks to ensure that the exception handler in Figure 4 has run. It then checks whether between 900ms and two seconds elapsed in response to the routine's 1 second sleep split into 10ms increments. Sandboxes using code like that available in [2] for example, would have run afoul of this particular sandbox check.

Obfuscation of Sandbox Detection via Skipping of Allocation of Critical Buffers

When Parasite HTTP actually does detect a sandbox, it attempts to hide this fact from any observers. It does not simply exit or throw an error, instead making it difficult for researchers to determine why the malware did not run properly and crashed. In the screenshot below, we can see how Parasite HTTP uses its sandbox detection in a clever way to result in a later crash on attempting to use a buffer whose allocation was skipped:

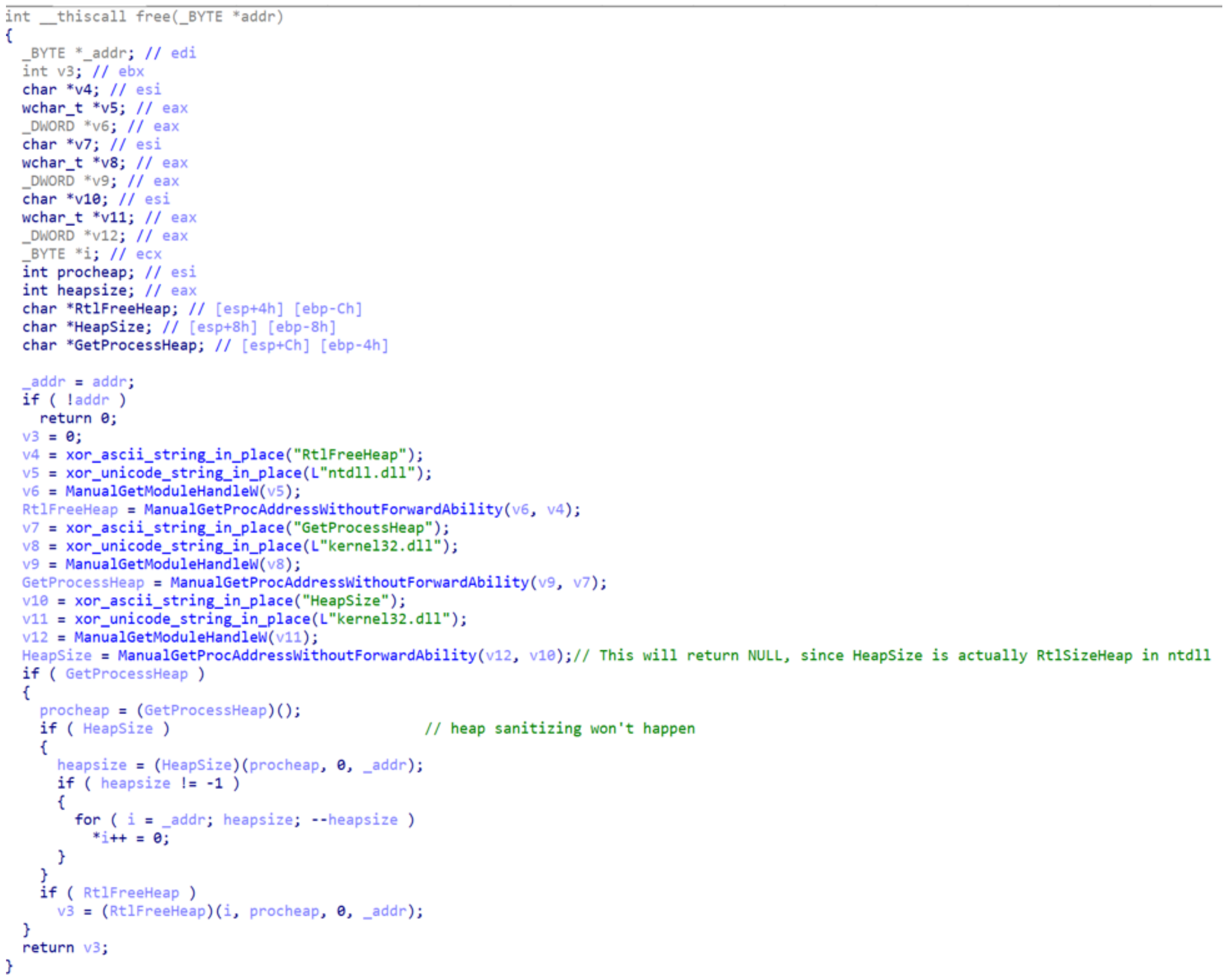


```
void __cdecl CheckForSandboxAndAllocateBuffers()
{
  unsigned int len; // ecx
  unsigned __int8 *p; // eax
  char *apiname; // esi
  wchar_t *dllname; // eax
  FARPROC InitializeCriticalSectionAndSpinCount; // eax

  if ( !IsSandboxBasedOnTimingAndDebugging() )
  {
    len = sizeof(GlobalStruct);
    p = &GStruct;
    do
    {
      *p++ = 0;
      --len;
    }
    while ( len );
    apiname = strdup_xored_string(&InitializeCriticalSectionAndSpinCount_string);
    dllname = xor_unicode_string_in_place(L"Kernel32.dll");
    InitializeCriticalSectionAndSpinCount = ResolveAPI(dllname, apiname);
    if ( (InitializeCriticalSectionAndSpinCount)(&GStruct.BufferLock, 1024) )
    {
      GStruct.C2Host = malloc(1024);
      GStruct.RC4Key1 = malloc(66);
      GStruct.RC4Key2 = malloc(66);
      GStruct.RC4Key3 = malloc(66);
    }
  }
}
```

Figure 5: Code snippet showing mechanism by which critical buffers required for malware functioning are not allocated if a sandbox is detected

Heap Clearing Bug

Parasite HTTP also contains a bug caused by its manual implementation of a GetProcAddress API that results in the clearing code not executing. This can be seen in the following commented decompilation:



```
int __thiscall free(_BYTE *addr)
{
  _BYTE *_addr; // edi
  int v3; // ebx
  char *v4; // esi
  wchar_t *v5; // eax
  _DWORD *v6; // eax
  char *v7; // esi
  wchar_t *v8; // eax
  _DWORD *v9; // eax
  char *v10; // esi
  wchar_t *v11; // eax
  _DWORD *v12; // eax
  _BYTE *i; // ecx
  int procheap; // esi
  int heapsize; // eax
  char *RtlFreeHeap; // [esp+4h] [ebp-Ch]
  char *HeapSize; // [esp+8h] [ebp-8h]
  char *GetProcessHeap; // [esp+Ch] [ebp-4h]

  _addr = addr;
  if ( !addr )
    return 0;
  v3 = 0;
  v4 = xor_ascii_string_in_place("RtlFreeHeap");
  v5 = xor_unicode_string_in_place(L"ntdll.dll");
  v6 = ManualGetModuleHandleW(v5);
  RtlFreeHeap = ManualGetProcAddressWithoutForwardAbility(v6, v4);
  v7 = xor_ascii_string_in_place("GetProcessHeap");
  v8 = xor_unicode_string_in_place(L"kernel32.dll");
  v9 = ManualGetModuleHandleW(v8);
  GetProcessHeap = ManualGetProcAddressWithoutForwardAbility(v9, v7);
  v10 = xor_ascii_string_in_place("HeapSize");
  v11 = xor_unicode_string_in_place(L"kernel32.dll");
  v12 = ManualGetModuleHandleW(v11);
  HeapSize = ManualGetProcAddressWithoutForwardAbility(v12, v10);// This will return NULL, since HeapSize is actually RtlSizeHeap in ntdll
  if ( GetProcessHeap )
  {
    procheap = (GetProcessHeap)();
    if ( HeapSize )                              // heap sanitizing won't happen
    {
      heapsize = (HeapSize)(procheap, 0, _addr);
      if ( heapsize != -1 )
      {
        for ( i = _addr; heapsize; --heapsize )
          *i++ = 0;
      }
    }
    if ( RtlFreeHeap )
      v3 = (RtlFreeHeap)(i, procheap, 0, _addr);
  }
  return v3;
}
```

Figure 6: Commented decompilation showing a bug in Parasite HTTP

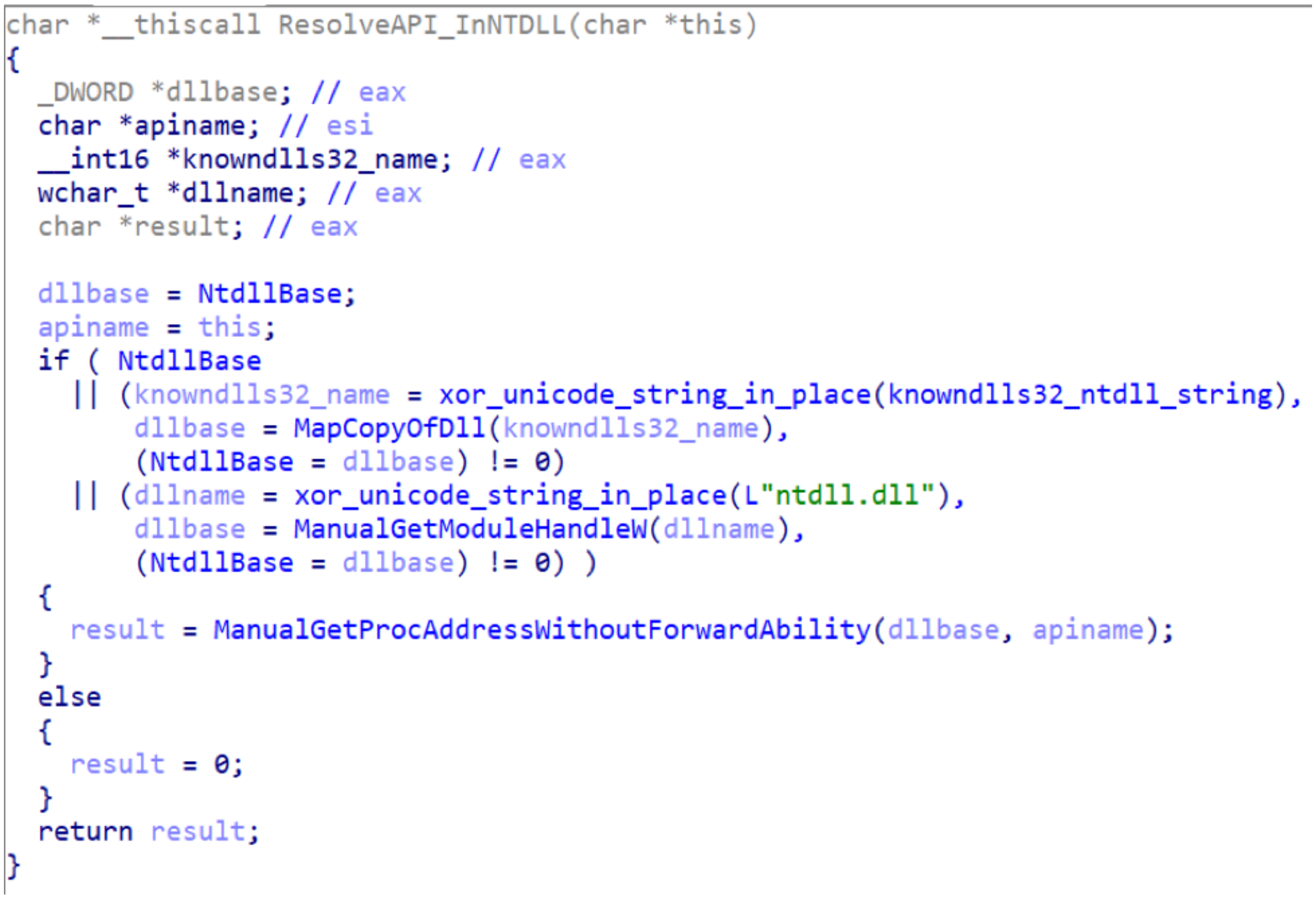
The malware attempts to resolve a function named HeapSize to its associated address within kernel32.dll. However, its manual GetProcAddress function lacks support for resolving forwarded exports. In this case, since HeapSize is in fact a forwarded export to NTDLL's RtlSizeHeap function, the function will return NULL and HeapSize and the associated clearing will never be called. This is easily confirmed by monitoring calls to RtlFreeHeap.

Use of Researcher Code from Github for Detecting Sandbox Hooking via Write Watches

Parasite HTTP adapts code from a public repository [3] for its own sandbox detection purposes. The code is copied verbatim, with the API resolution replaced with its own internal code, the prints removed, and the file and environment variable names generated randomly. Of note is that the GetWriteWatch API in this case detects more than just writes. Due to demand paging in Windows, the first access, whether read or write, to the allocation created will result in the page table entries being instantiated and counted as a "write" for the purposes of GetWriteWatch. Since the APIs being called with invalid arguments would never read or write to the allocated buffer, hooks that read from the buffer prematurely or even in the case of failure of the API, as in [4], would fail this sandbox detection.

Remapping of NTDLL via KnownDlls32\ntdll.dll for Hook Evasion

On Microsoft Windows, versions 7 and newer that have KnownDlls functionality, Parasite HTTP resolves certain critical APIs by using a DLL remapping technique that while previously documented, has not, to our knowledge, been used recently in other major malware families. Malware behavior hidden by this technique include process injection and the Poweliks technique of including a NUL character in registry value names. The malware preserves support for older versions of Windows by falling back to using the existing copy of NTDLL loaded in memory. This behavior can be seen in the following decompilation listing:



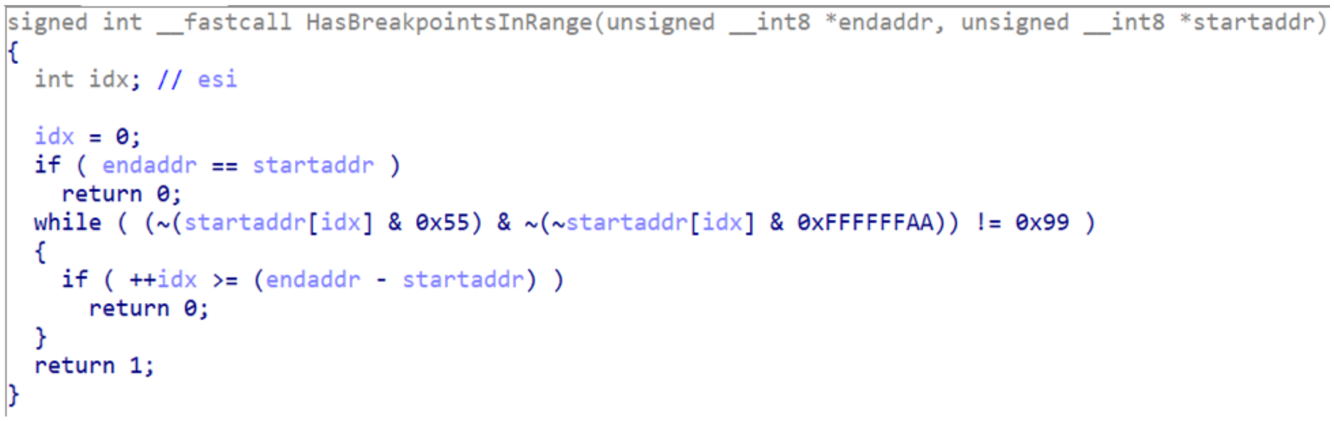
```
char *__thiscall ResolveAPI_InNTDLL(char *this)
{
  _DWORD *dllbase; // eax
  char *apiname; // esi
  __int16 *knowndlls32_name; // eax
  wchar_t *dllname; // eax
  char *result; // eax

  dllbase = NtdllBase;
  apiname = this;
  if ( NtdllBase
    || (knowndlls32_name = xor_unicode_string_in_place(knowndlls32_ntdll_string),
        dllbase = MapCopyOfDll(knowndlls32_name),
        (NtdllBase = dllbase) != 0)
    || (dllname = xor_unicode_string_in_place(L"ntdll.dll"),
        dllbase = ManualGetModuleHandleW(dllname),
        (NtdllBase = dllbase) != 0) )
  {
    result = ManualGetProcAddressWithoutForwardAbility(dllbase, apiname);
  }
  else
  {
    result = 0;
  }
  return result;
}
```

Figure 7: Code snippet showing NTDLL remapping

Mapping the new copy of NTDLL effectively provides it with a copy free of any hooks placed on the initial NTDLL mapping, rendering its thread injection and registry modifications invisible to most userland hooking implementations. Further, since this mapping is accomplished with NtOpenSection and NtMapViewOfSection, it will not involve the typical calls to filesystem APIs used by other variants of the technique to achieve the same goal.

Obfuscated Checking for Breakpoints within a Critical Function of the Malware

Parasite HTTP includes an obfuscated check for debugger breakpoints within a range of its own code. This functionality is only used in one location to check a single function in the malware that calls out to the sandbox detection based on GetWriteWatch checks (Figure 8):



```
signed int __fastcall HasBreakpointsInRange(unsigned __int8 *endaddr, unsigned __int8 *startaddr)
{
  int idx; // esi

  idx = 0;
  if ( endaddr == startaddr )
    return 0;
  while ( (~(startaddr[idx] & 0x55) & ~(~startaddr[idx] & 0xFFFFFFAA)) != 0x99 )
  {
    if ( ++idx >= (endaddr - startaddr) )
      return 0;
  }
  return 1;
}
```

Figure 8: Obfuscated check for debugger breakpoints

This code is also copied from [1] with the malware having implemented the added “level of indirection” mentioned in the code comment. It is worth noting that this technique is naive and unreliable long-term over arbitrary code, as unintentional 0xcc bytes can be found in a simple byte-by-byte scan of code through certain instruction encodings, local stack frame offsets, relative references, indirect addresses, or immediate constants.

Extension of Themida/Formbook Technique for Unhooking APIs in NTDLL/Kernel32/Kernelbase

In its initial process, Parasite HTTP removes hooks on the aforementioned DLLs by reading them in from disk and comparing the first 5 bytes of each exported function to that present in the currently mapped version in memory. Though this technique is naive in its implementation, not making use of any instruction decoder and limiting itself to 5 hardcoded bytes, it is effective in practice. Consider the case of a sandbox using an indirect jump (6 bytes) for its hooks -- the malware will restore only the first 5 bytes to the original, leaving the final byte of the hook in place, most likely resulting in a crash upon its execution.

Conclusion

Threat actors and malware authors continuously innovate in their efforts to evade defenses and improve infection rates. Parasite HTTP provides numerous examples of state-of-the-art techniques used to avoid detection in sandboxes and via automated anti-malware systems. For consumers, organizations, and defenders, this represents the latest escalation in an ongoing malware arms race that extends even to commodity malware like Parasite. While we have currently only observed Parasite HTTP in a small campaign, we expect to see features like those used in Parasite continue to propagate across other malware variants.

References

[1] https://github.com/LordNoteworthy/al-khaser/blob/master/al-khaser/Anti%20Debug/Interrupt_3.cpp

[2] https://github.com/spender-sandbox/cuckoomon-modified/blob/MSVC/hook_sleep.c#L122

[3] https://github.com/LordNoteworthy/al-khaser/blob/master/al-khaser/Anti%20Debug/WriteWatch.cpp

[4] https://github.com/spender-sandbox/cuckoomon-modified/blob/MSVC/hook_thread.c#L232

Indicators of Compromise (IOCs)

IOC	IOC Type	Description
6479a901a17830de31153cb0c9f0f7e8bb9a6c00747423adc4d5ca1b347268dc	SHA256	Macro Document
hxxp://dboxhost[.]tk/moz/bza.exe	URL	Document Payload (Parasite HTTP)
b52706530d7b56599834615357e8bbc1f5bed669001c06830029784eb4669518	SHA256	Parasite HTTP
xetrodep[.]top	Domain	Parasite HTTP C&C
jekoslo[.]space	Domain	Parasite HTTP C&C
befrodet[.]top	Domain	Parasite HTTP C&C

ET and ETPRO Suricata/Snort/ClamAV Signatures

2831834 || ETPRO TROJAN Parasite HTTP Checkin

Subscribe to the Proofpoint Blog