Daxin Backdoor: In-Depth Analysis, Part Two

By About the Author

Archived: 2026-04-06 01:37:29 UTC

This is the concluding part of our in-depth analysis of Backdoor. Daxin, advanced malware that is being used by a China-linked espionage group.

In this blog, we will analyze the communications and network features of the malware.

Communications protocol

In our previous blog, we set up a lab consisting of four separate networks and five machines. Some of the machines had two network interfaces to communicate with different networks, but all packet forwarding functionality was disabled. Each machine ran various network services that were reachable from its neighbors only.


Figure 1. Test setup to illustrate Daxin’s backdoor capabilities.

In this section we will dissect the network traffic between two backdoor instances running on the separate computers “Alice-PC” and “Bob-PC”. The traffic was initiated by the Daxin backdoor running on “Alice-PC” when it was instructed to create a communication channel to “Dave-PC” passing via two intermediate nodes, “Bob-PC” and “Charlie-PC”, as described previously.


Figure 2. Wireshark capture of traffic between two backdoor instances. The screenshot and examples below are reused from a private report prepared by us that discussed an earlier sample, so certain details may not match.

Using Wireshark, we captured traffic between two backdoor instances, one running on “Alice-PC” and the other on “Bob-PC”, as shown in Figure 2.

Starting with the key exchange, all backdoor communication is carried out by exchanging messages that follow the same underlying format:

```
 import struct def dissect_message(message): magic, kind, unknown_03, total_length = struct.unpack("
<HBBI", message[: 8]) assert total_length == len(message) print(f"0000 magic = {magic:04x}")
print(f"0002 kind = {kind:02x}") print(f"0003 unknown_03 = {unknown_03:02x}") print(f"0004
total_length = {total_length:08x}") return magic, kind, message[8: ]
```

The *magic* value is always 0x9910 or 0x9911.

The *kind* value identifies the state transition during key exchange. Then, once the encrypted communication channel is established, it encodes the purpose of each message and determines the formatting of the data that follows the fixed-size header.

The initial message of the key exchange in the Wireshark capture is not encrypted:

```
class Session: pass def decode_key_exchange_1_message(tcp_dump): message = bytes.fromhex(tcp_dump) magic, kind, message_body = dissect_message(message) assert magic == 0x9910 assert kind in [0x10, 0x11] print(f"0008 message_body (unused) = {message_body}") return Session() my_session = decode_key_exchange_1_message("""  10 99 11 00 08 00 00 00 """)
```

It can be decoded as follows:

```
0000 magic = 9910 0002 kind = 11 0003 unknown_03 = 00 0004 total_length = 00000008 0008 message_body (unused) = b''
```

The fields *magic* and *kind* correspond to the first three bytes of TCP data, 0x10 0x99 0x11. On the target computer, in case it is infected with a copy of the malicious driver, this sequence causes the TCP connection to be hijacked, as explained in part one of this blog series.

The target checks that the received message is valid according to the session state machine, ensuring that *magic* is the expected constant 0x9910 and *kind* matches any of two supported values: 0x10 or 0x11. Next, it generates a nonce to use when encrypting any future incoming messages. Finally, it sends a response message with the nonce, its own details, and the information about the infected machine.

Parts of the response message are encrypted using a combination of the following algorithms:

```
import hashlib import itertools def rc4_variant(key): """Variant of RC4 with modified initial value of j in PRGA. The initial value of j in PRGA is from the final KSA operation and may not be zero. """ S = bytearray(range(0x100)) j = 0 cycled_key = itertools.cycle(key) for i in range(0x100): j = (j + S[i] + next(cycled_key)) & 0xff S[i], S[j] = S[j], S[i] i = 0 # skipping j reinitialization while True: i = (i + 1) & 0xff j = (j + S[i]) & 0xff S[i], S[j] = S[j], S[i] K = S[(S[i] + S[j]) & 0xff] yield K def rol(value, count, width=8): mask = (1 << width) - 1 return (((value << count) & mask) | ((value & mask) >> (width - count)) ) def xor_crypt(data, key_stream): return bytes([byte ^ next(key_stream) for byte in data]) def derive_key(nonce): md5 = hashlib.md5() md5.update([REDACTED]) md5.update(nonce) rc4_variant_key = bytearray() for byte in md5.digest(): rc4_variant_key.append([REDACTED]) rc4_variant_stream = rc4_variant(rc4_variant_key) return xor_crypt(nonce, rc4_variant_stream)
```

The details of this response message are as follows:

```
from socket import inet_ntoa def decode_key_exchange_2_message(session, tcp_dump): message = bytes.fromhex(tcp_dump) magic, kind, message_body = dissect_message(message) assert magic == 0x9910 assert kind == 0x12 assert 0x114 <= len(message_body) unknown_00 = message_body[: 0x10] session.target_build = int.from_bytes(message_body[0x10: 0x13], "little") session.target_version = int.from_bytes(message_body[0x13: 0x14], "little") encrypted_target_nonce = message_body[0x14: 0x94]
```

```
encrypted_information = message_body[0x94: 0xac] unused = message_body[0xac: ] cycled_key = itertools.cycle(b"\x10\x99") target_nonce = xor_crypt(encrypted_target_nonce, cycled_key) key_stream = rc4_variant(target_nonce) information = xor_crypt(encrypted_information, key_stream) recognised_login = information[: 0x10] initiator_ip_addr = information[0x10: 0x14] netmask = information[0x14: ] print(f"0008 message_body:") print(f"0008 unknown_00 (marker?)= {unknown_00}") print(f"0018 target_build? = {session.target_build}") print(f"001b target_version? = {session.target_version}") print(f"001c target_nonce = {target_nonce.hex()}") print(f"009c information:") print(f"009c recognised_login = {recognised_login}") print(f"00ac initiator_ip_addr = {inet_ntoa(initiator_ip_addr)}") print(f"00b0 netmask = {inet_ntoa(netmask)}") print(f"00b4 unused = {unused}") session.target_key = derive_key(target_nonce) session.target_key_stream = itertools.cycle(session.target_key) decode_key_exchange_2_message(my_session, """ 10 99 12 00 1c 01 00 00 78 72 74 00 83 fa ff ff 00 00 00 00 00 00 00 00 75 07 00 13 89 5c 69 be 7f 09 60 ed 3d 3d 9c c5 ef 55 67 29 14 67 70 2d ef a7 5e bb 5f 59 a4 e8 31 d4 55 cf f4 7e 19 15 ed 29 a1 48 ac 91 38 a8 9f 6f 7a 1f 66 50 89 d9 a7 66 5e fb c3 fc 3d 54 a8 19 ac 58 28 b5 65 38 aa 1a 09 6f 20 99 58 2e 59 7d 5f 51 f7 40 51 59 c3 4f 59 76 a0 db 2e 39 3d c5 68 02 6f 43 f2 77 1f 49 41 74 7f 4d 04 e1 83 f9 dd e5 2e e1 6f 25 aa 15 f7 4b dc 06 00 e9 f7 30 32 75 82 7b 14 87 cb 40 2b 42 8e 54 5a dd c4 21 d2 b7 8f 88 25 02 95 e0 4d 4e 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 """)
```

The message includes the backdoor login that is recognized during the following key exchange step, and what looks like malware build and version numbers.

Looking at the decoded message, we find references to “XRT” and these reassemble the hardcoded Name “NDISXRPT” that we documented when discussing the *NdisRegisterProtocol()* call during driver initialization:

```
 0000 magic = 9910 0002 kind = 12 0003 unknown_03 = 00 0004 total_length = 0000011c 0008 message_body: 0008 unknown_00 (marker?) = b'xrt\x00\x83\xfa\xff\xff[REMOVED ZEROS]' 0018 target_build? = 1909 001b target_version? = 19 001c target_nonce = 99c57927[REMOVED FOR BREVITY]e7a922ec 009c information: 009c recognised_login = b'XRT[REMOVED ZEROS]' 00ac initiator_ip_addr = 10.0.2.1 00b0 netmask = 255.255.255.0 00b4 unused = b'[REMOVED ZEROS]'
```

The initiator responds with the backdoor login and hashed password:

```
 def decode_key_exchange_3_message(session, tcp_dump): encrypted_message = bytes.fromhex(tcp_dump) message = xor_crypt(encrypted_message, session.target_key_stream) magic, kind, message_body = dissect_message(message) assert magic == 0x9911 assert kind == 0x15 assert 0x94 <= len(message_body) login = message_body[: 0x10] password_hash = message_body[0x10: 0x14] initiator_nonce = message_body[0x14: 0x94] unused = message_body[0x94: ] print(f"0008 message_body:") print(f"0008 login = {login}") print(f"0018 password_hash = {password_hash.hex()}") print(f"001c initiator_nonce = {initiator_nonce.hex()}") print(f"009c unused = {unused}") session.initiator_key = derive_key(initiator_nonce) session.initiator_key_stream = itertools.cycle(session.initiator_key) md5 = hashlib.md5() md5.update([REDACTED]) md5.update(session.target_key[: 0x40])
```

```
md5.update(session.initiator_key[0x40: ]) if (session.target_build > 1410) and (session.target_version >= 16): md5.update([REDACTED]) session.shared_key = md5.digest()
decode_key_exchange_3_message(my_session, """ 26 dc d1 0c 6e d9 52 76 e4 7d 56 33 36 a7 a1 46 76 74 37 80 5f f5 89 69 xx xx xx xx 98 9f 16 9a 11 73 23 01 56 70 bd 13 fb a8 b6 8c bf 04 ae b1 dc b8 22 44 da 1b bb c0 59 87 c3 0f 55 66 89 ae 14 84 70 89 7d 6e a0 28 3e ff 8e 7c da 99 a7 00 ad 1b c7 63 72 60 c7 4a 09 df 4c fb d8 b2 da 56 b4 de 71 3b 7e a5 c0 d4 28 bd 55 5c 2c 23 42 51 76 0f ad 5d 8e eb c6 f9 05 38 81 42 07 c6 5c 5f a0 22 94 b0 9f f0 2e 6d 5f 7e ab d4 fa 55 4d a8 ff 0a 09 d3 d7 cf ad f3 74 fb 88 48 """)
```

It also includes its own nonce, like this:

```
0000 magic = 9911 0002 kind = 15 0003 unknown_03 = 00 0004 total_length = 0000009c 0008 message_body: 0008 login = b'XRT[REMOVED ZEROS]' 0018 password_hash = [REDACTED] 001c initiator_nonce = b0bf1c98[REMOVED FOR BREVITY]0bee8fff 009c unused = b''
```

At this point, the peers exchange their nonces and compute two transport keys. Each transport key is used in stream mode to encrypt the TCP half-stream directed towards the side that generated the corresponding nonce.

The peers also combined their nonces into the shared key. This shared key will be used to encrypt the body of each exchanged message, reusing the same key every time.

The final key exchange message confirms that the initiator was successfully authenticated and the backdoor is ready to process instructions:

```
def dissect_encrypted_message(transport_key_stream, shared_key, encrypted_message): message = xor_crypt(encrypted_message, transport_key_stream) magic, kind, encrypted_message_body = dissect_message(message) key_stream = rc4_variant(shared_key) message_body = xor_crypt(encrypted_message_body, key_stream) return magic, kind, message_body def decode_key_exchange_4_message(session, tcp_dump): encrypted_message = bytes.fromhex(tcp_dump) magic, kind, message_body = dissect_encrypted_message(session.initiator_key_stream, session.shared_key, encrypted_message) assert magic == 0x9911 assert kind == 0x16 print(f"0008 message_body (unused) = {message_body}") decode_key_exchange_4_message(my_session, """ 44 4a 18 ce 90 a5 67 2f """)
```

It can be decoded as follows:

```
0000 magic = 9911 0002 kind = 16 0003 unknown_03 = 00 0004 total_length = 00000008 0008 message_body (unused) = b''
```

For the messages that follow, the *kind* field encodes the message purpose. This determines the formatting of the message body. For example, the backdoor instruction to set up new connectivity across multiple malicious nodes uses *kind* value 6 with the following message body structure:

```
def format_kind_06_message_body(message_body): number_of_nodes = int.from_bytes(message_body[: 2], "little") remaining_to_connect = int.from_bytes(message_body[2: 4], "little") print(f"0008 message_body:") print(f"0008 number_of_nodes = {number_of_nodes}") print(f"000a remaining_to_connect = {remaining_to_connect}") offset = 4 for index in range(number_of_nodes): ip_addr =
```

```
message_body[offset: offset + 4][:: -1] port = int.from_bytes(message_body[offset + 4: offset + 6], "little") login = message_body[offset + 6: offset + 0x16] password = message_body[offset + 0x16: offset + 0x38] comment = "" if index + remaining_to_connect == number_of_nodes: comment = " (HEAD)" print(f"{offset + 0x08:04x} node #{index + 1}{comment}:") print(f"{offset + 0x08:04x} ip_addr = {inet_ntoa(ip_addr)}") print(f"{offset + 0x0c:04x} port = {port}") print(f"{offset + 0x0e:04x} login = {login}") print(f"{offset + 0x1e:04x} password = {password}") offset += 0x38 unused = message_body[offset: ] print(f"{offset + 0x08:04x} unused = {unused}")
```

We could continue to decrypt all the backdoor communication that follows:

```
message_body_formatters = { 0x06: format_kind_06_message_body, } def format_message_body(message_body): print(f"0008 message_body = {message_body}") def decode_encrypted_initiator_message(session, tcp_dump): encrypted_message = bytes.fromhex(tcp_dump) magic, kind, message_body = dissect_encrypted_message(session.target_key_stream, session.shared_key, encrypted_message) message_body_formatters.get(kind, format_message_body)(message_body) def decode_encrypted_target_message(session, tcp_dump): encrypted_message = bytes.fromhex(tcp_dump) magic, kind, message_body = dissect_encrypted_message(session.initiator_key_stream, session.shared_key, encrypted_message) message_body_formatters.get(kind, format_message_body)(message_body)
```

For example, the next message in the captured network traffic is:

```
decode_encrypted_initiator_message(my_session, """ 39 b9 0c 02 7b f8 d1 a4 b7 a3 8f 4b 15 f4 33 33 a2 be aa 75 14 46 8f 25 62 7b fa 22 01 24 6a ee 36 c0 xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx d0 77 c6 16 35 bd 3a 39 6d df 9a 8b cb de 6a a0 8d e7 f4 e7 e6 ae xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx 72 fc a0 2a cd 21 04 57 41 e8 17 68 0a f4 de 18 6a 80 99 39 f7 b6 xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx """)
```

This instructs the backdoor to set up remaining connectivity across malicious nodes as explained in the “Backdoor capabilities” section in our previous blog:

```
0000 magic = 9911 0002 kind = 06 0003 unknown_03 = 00 0004 total_length = 000000b4 0008 message_body: 0008 number_of_nodes = 3 000a remaining_to_connect = 2 000c node #1: 000c ip_addr = 10.0.2.2 0010 port = 80 0012 login = b'XRT[REMOVED ZEROS]' 0022 password = b'[REDACTED]' 0044 node #2 (HEAD): 0044 ip_addr = 10.0.3.3 0048 port = 80 004a login = b'XRT[REMOVED ZEROS]' 005a password = b'[REDACTED]' 007c node #3: 007c ip_addr = 10.0.4.4 0080 port = 80 0082 login = b'XRT[REMOVED ZEROS]' 0092 password = b'[REDACTED]' 00b4 unused = b''
```

The most interesting observation about encryption is that Daxin supports two methods for computing the shared key during the key exchange. To select which of the two methods to use, the initiator examines the target message, comparing what looks like build and version numbers against certain hardcoded constants. This could be in order to facilitate upgrading the malicious network in the field without disruption.

Additionally, the current key exchange implementation involves additional obfuscation that is not present in some older samples. It is possible that the attacker was forced to change the algorithm and decided to implement

additional measures to protect the details of new logic.

An alternative explanation is that different teams within the attacker organization were sharing the same codebase, where one of these teams implemented the alternative key exchange method and related obfuscations to mitigate against potential compromise due to the other team's activity, while still sharing some of the communication infrastructure.

External communication

The communications protocol documented in the previous section is how two backdoor instances communicate with each other.

On top of that, the analyzed sample also supports two additional communication methods. These additional methods are well suited for crossing the perimeter of the target organization.

HTTP

One of these additional communication methods uses HTTP messages to encapsulate backdoor communications.

To demonstrate this, we implemented our own client to interact with the backdoor using this method. Our client communicated with the backdoor instance running on "Alice-PC" over HTTP to control a set of infected machines, as discussed in the previous two sections.

On the target computer, in case it is infected with a copy of Daxin, the first HTTP request causes the TCP connection to be hijacked due to the malicious packet filter triggering on the HTTP "POST" method string with URI substring "756981520337" as explained in the "Networking" section of our previous blog.

Daxin then parses HTTP request headers and extracts the request body. The request body is then interpreted using the same logic as already described in the "Communications protocol" section.

The reverse communication is then encapsulated as the HTTP response body. When generating the HTTP response, the malicious driver includes "SID" cookie. The value of "SID" cookie is then used when constructing the URI for the subsequent HTTP request.

"HOST" connectivity

The malicious driver can also be configured to communicate with a remote TCP server for command and control. It then periodically connects to the remote server, performs a handshake that is unique to this connectivity method, and then starts backdoor communication.

This connectivity method is controlled with persistent configuration that can be updated by the remote attacker, as explained in the "Backdoor capabilities" section of our previous blog.

To obtain the details of the TCP server to connect to, Daxin checks the value of the "HOST" configuration item. In case the value starts with "http://", the TCP server details are retrieved from the remote web server, as described below. Otherwise, the configuration value is interpreted as the TCP server address and port.

In order to retrieve the TCP server details from the remote web server, the analyzed sample contacts the provided URL and scans the received HTTP response, including HTTP headers, for magic strings. Whenever it finds “f8xD4C01” followed later by “d7C6x12B”, it attempts to interpret any data immediately following the first marker as a hexadecimal string. The data obtained by decoding the hexadecimal string is then decrypted using the following algorithm:

```
def decrypt(data): return bytes([((byte - 0x7d) & 0xff) ^ 0x49 for byte in data])
```

The decrypted data are interpreted as the TCP server address and port to use.

Whenever the analyzed sample connects to the TCP server, it sends the following sequence of bytes as its handshake:

```
def serialize_client_handshake(tags_value): magic = b"\xA8\xB0\x13\x7C\x00\x2C\x13\xBA" build = 1909 return magic + struct.pack("<HH", build, len(tags_value)) + tags_value
```

The *tags_value* parameter is the value of the “TAGS” configuration item. We suspect that the remote server uses the *tags_value* for tracking specific infections.

The analyzed sample then expects to receive the following hardcoded sequence of bytes from the remote server:

```
def serialize_server_handshake(): return b"\xA8\xB0\x13\x7C\x45\x1B\xAC\xC0"
```

This should be followed by the usual key exchange as described in the “Communication protocol” section, where the remote server acts as initiator.

Conclusion

This concludes the second and final part of our technical analysis of Backdoor.Daxin.
