

Cobalt Strike Process Injection

0x01 Intro

Here are my thoughts on process injection and share some technical details about Cobalt Strike's process injection, as well as some of the red team attack techniques you may want to know.

0x02 Injection Function

Cobalt Strike currently provides process injection functions in some scenarios. The most common is to directly inject payload into a new process. This function can be executed through various sessions that you have obtained, such as [Artifact Kit](#) , [Applet Kit](#) and [Resource Kit](#) . This article will focus on Cobalt Strike's process injection in Beacon sessions.

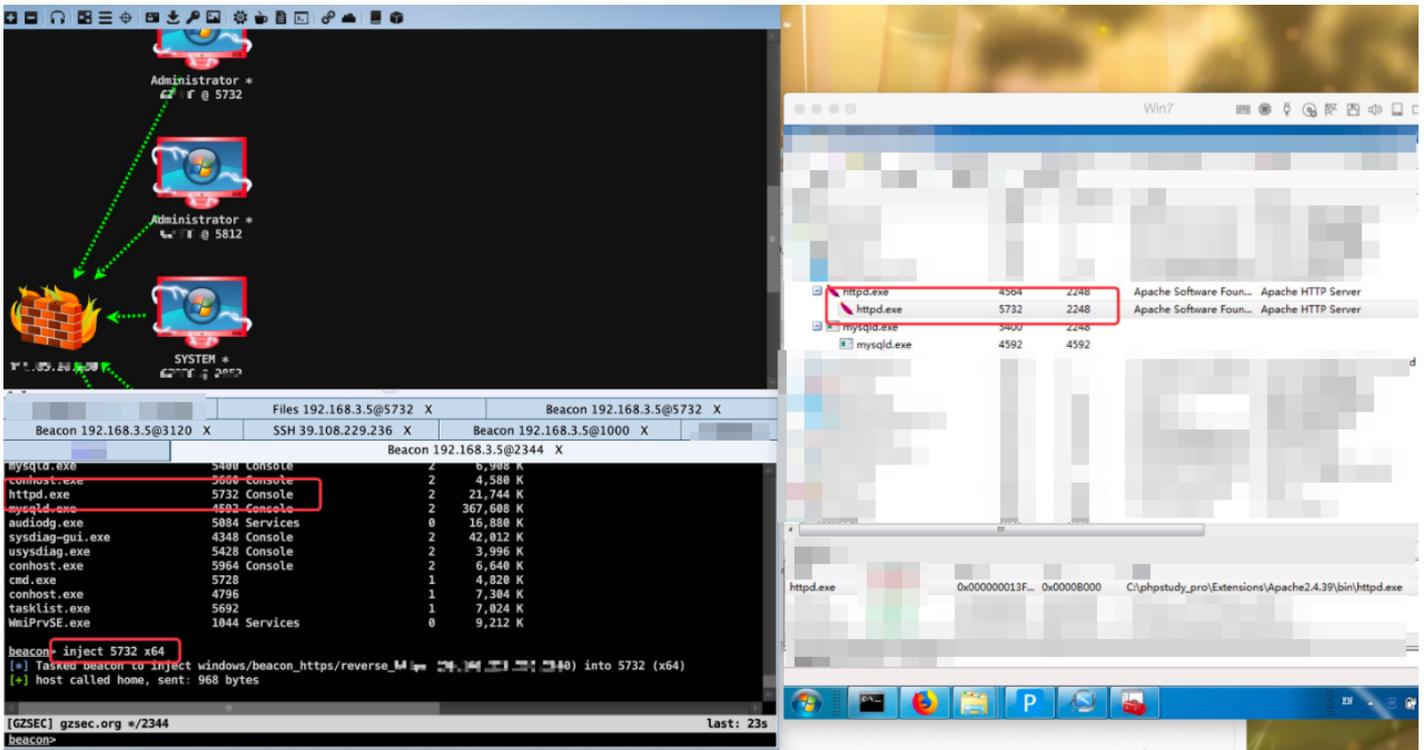
The `shinject` command injects code into any remote process, some built-in *post-exploitation* modules can also be injected to a particular remote process through the tool. Cobalt Strike did this because injecting shellcode into a new session would be safer than migrating the session directly to another C2.

(Probably the reason is that if the new session is not pulled up, it will be embarrassing if the original session has dropped.)

Therefore, Cobalt Strike *post-exploitation* will start a temporary process when it is executed, and inject the DLL file corresponding to the payload into the process, and confirm the result of the injection by retrieving the named pipe. Of course, this is just a special case of process injection. In this way, we can safely operate the main thread of these temporary processes without worrying about operation errors that cause the program to crash and result in loss of permissions. This is a very important detail to understand when learning to use Cobalt Strike injection process.

The first parameter of the inject command mentioned in the original text is the PID of the target program to be injected, and the second parameter is the architecture of the target program. If not filled, the default is x86.

```
inject 5732 x64
```



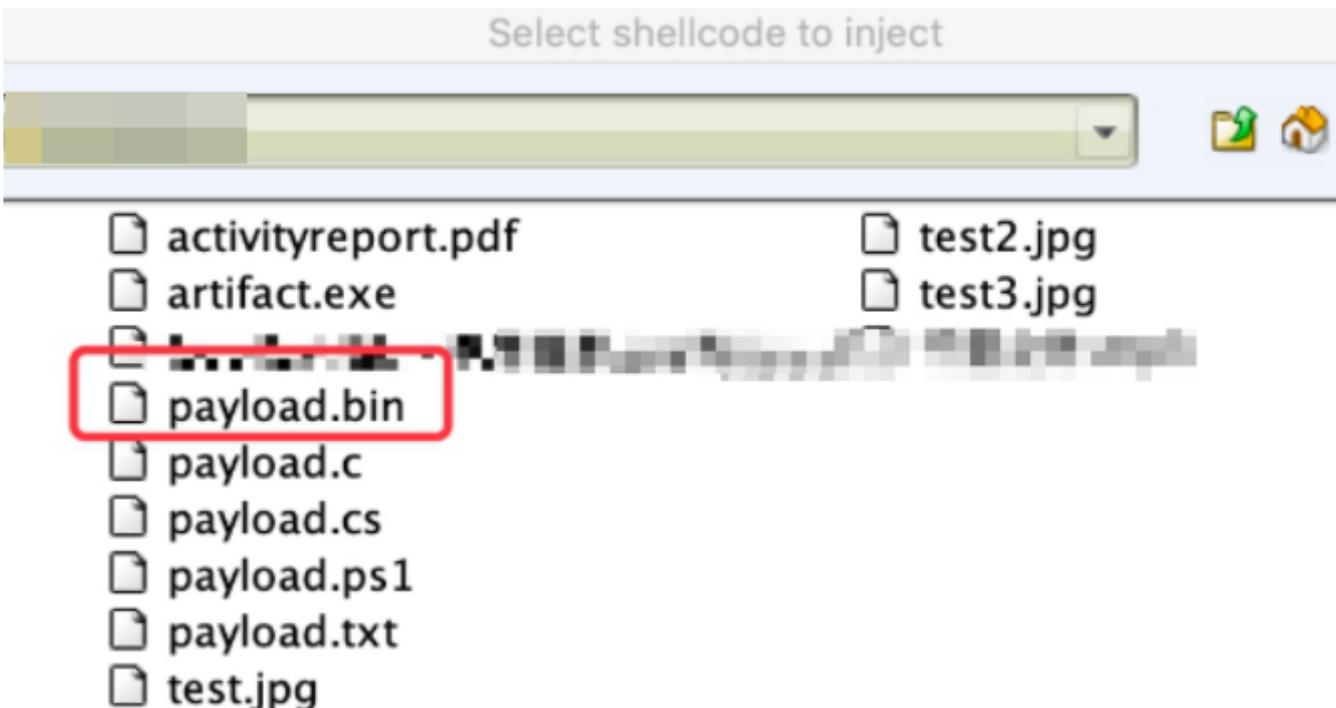
```

09:48:01 *** initial beacon from gzsec.org *@192.168.3.5 (■■■■■)
10:31:14 *** initial beacon from Administrator *@192.168.3.5 (■■■■■)
10:38:05 *** initial beacon from Administrator *@192.168.3.5 (■■■■■)
10:44:01 *** initial beacon from Administrator *@192.168.3.5 (■■■■■)
10:46:08 *** initial beacon from SYSTEM *@192.168.3.5 (■■■■■)

```

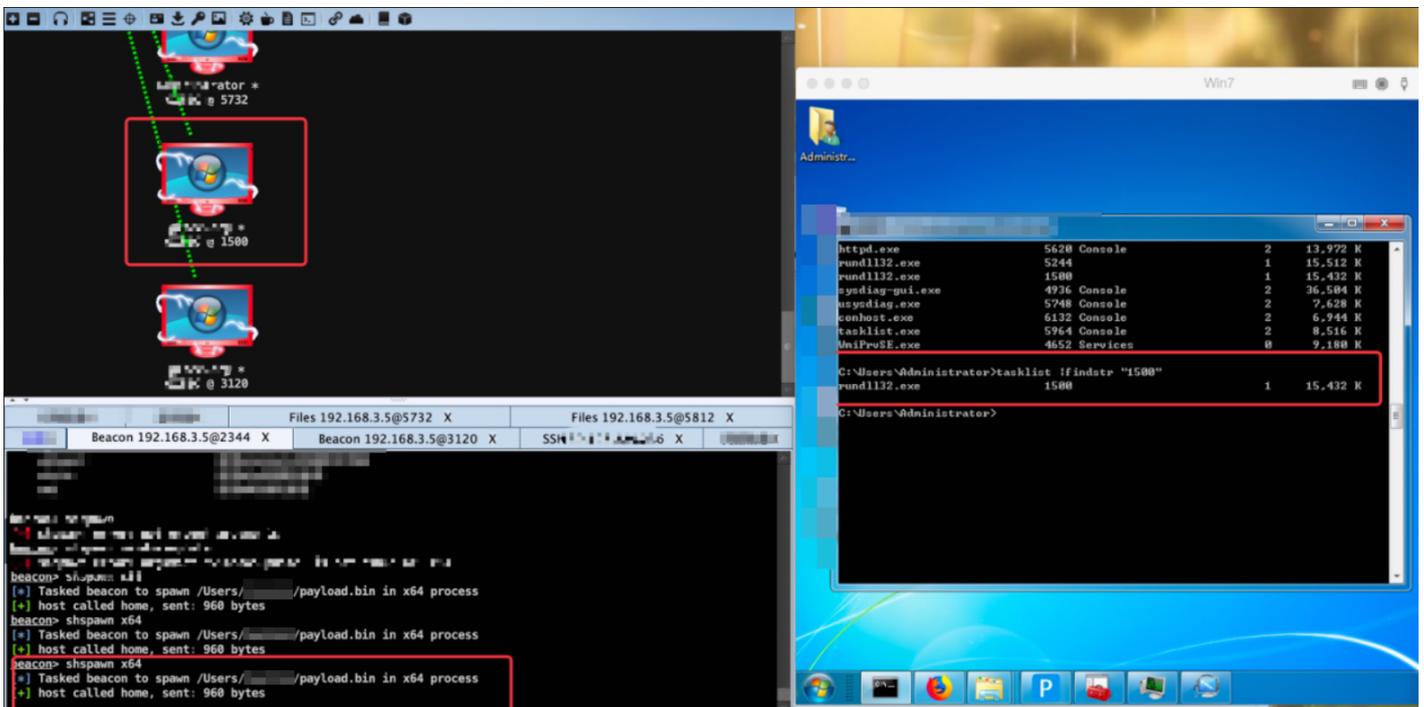
The parameter writing method of `shinject` is the same as that of `inject`. If the third parameter is not written, you will be prompted to select a shellcode file. Pay attention to the bin format payload that needs to be generated.

```
shinject 5732 x64 /xxx.bin
```



In addition to the two beacon commands mentioned in the original text, in fact, there is also a `shspawn`. Its role is to start a process and inject shellcode into it. The parameters only need to select the program architecture.

```
shspawn x64 /xxx.bin
```



As shown in the figure, the payload is injected into the `rundll32.exe` program. This method is much more stable than the first two, and you are not afraid of crashing the program.

0x03 injection process

The `process-inject` block in Cobalt Strike's Malleable C2 configuration file is where the configuration process is injected:

```
process-inject {
    # set remote memory allocation technique
    set allocator "NtMapViewOfSection";

    # shape the content and properties of what we will inject
    set min_alloc "16384";
    set userwx    "false";

    transform-x86 {
        prepend "\x90";
    }

    transform-x64 {
        prepend "\x90";
    }

    # specify how we execute code in the remote process
    execute {
        CreateThread "ntdll! RtlUserThreadStart";
        CreateThread;
        NtQueueApcThread-s;
        CreateRemoteThread;
        RtlCreateUserThread;
    }
}
```

The execution flow of this code is roughly as follows:

1. Open the handle of the remote process.
2. Allocate memory in remote processes.
3. Copy the shellcode to the remote process.
4. Execute shellcode in the remote process.

Step 1: Distribute and copy data to the remote host

The first step exists but daily development does not pay much attention. If we start a temporary process (such as a call *post-exploitation*); that is, we already have a handle to the remote process, at this time if we want to inject the code into the existing remote process ... [manual dog head], Cobalt Strike will use [OpenProcess](#) to solve this problem.

Step 2-3

Cobalt Strike provides two options for allocating memory and copying data into remote processes.

The first solution is the classic- `VirtualAllocEx` `WriteProcessMemoryPattern`, which is very common in attack tools. It is worth mentioning that this solution is also applicable to different process architectures, and the application of process injection is not limited to the injection of x64 target processes. This means that a good solution needs to take into account the different extreme situations that occur (for example, x86-> x64, or x64-> x86, etc.). This makes it `VirtualAllocEx` a relatively reliable choice, and Cobalt Strike's default solution is also his. If you want to directly specify this mode, you can set `process-inject` the `-> allocator` option `VirtualAllocEx` to.

The second solution provided by Cobalt Strike is `CreateFileMapping` `-> MapViewOfFile` `-> NtMapViewOfSection` mode. This solution will first create a mapping file that supports the Windows system, and then map the view of the mapping file to the current process. Then Cobalt Strike will copy the injected data to the memory associated with the view and `NtMapViewOfSection` call our remote process. The same mapping file available in. To use this scheme `process-inject` `-> allocator` set `NtMapViewOfSection` to the disadvantage of this scheme is only for `x86` `-> x86` and `x64` `-> x64`, related to the cross-architecture injection when Cobalt Strike will automatically switch back to `VirtualAllocEx` mode.

When `VirtualAllocEx` `> WriteProcessMemory` mode injection is subject to anti-soft defense? It is also a good choice to try this scheme instead. (It is very useful when killing software without detecting other methods of copying data to a remote process.)

Data conversion

The steps 2 and 3 mentioned above assume that everything is normal and the original data is copied to the injected data, which is almost impossible in the real environment. To this end, Cobalt Strike's process-inject adds the function of transforming and injecting data. The `min_alloc` option is the minimum size of the block that Beacon will allocate in the remote process, `start_rwx` and the `user_rwx` option is the initial Boolean value of the allocated memory and the final permission of the allocated memory. If you want to prohibit data from being readable, writable, and executable (`RWX`), please set these values to `false`. `transform-x86` and those that `transform-x64` support converting data to another architecture. If you need to add data in advance, make sure it is executable code for the corresponding architecture.

Note, many content signatures look for specific bytes at a fixed offset at the beginning of the

observable boundary. These checks occur in $O(1)$ time, which is conducive to $O(n)$ search. Excessive inspection and security technology may consume a lot of memory, and performance will be reduced accordingly.

Binary padding also affects `post-exploitation` the thread start address offset in Cobalt Strike. When Beacon injects a DLL into memory; its `ReflectiveLoader` starts the thread at the position where the function exported by the DLL should be. This offset is shown in the thread start address feature, and is looking for a specific *post-exploitation* DLL of `potential indicators`. The data before injection into the DLL will affect this offset. (It 's okay not to know about thread related things, I will talk about it next ...)

Step 4: code execution

Let's take a look at the subtle differences between different execution methods in Beacon:

CreateThread

`CreateThread` from the beginning. I think that `CreateThread` if it exists, should first appear in an execution block, this function only runs when it is limited to self-injection. Using `CreateThread` will start a thread pointing to the code you want your Beacon to run. But be careful, when you self-inject in this way, the thread you pull will have a starting address, which is not related to the module (by module I mean DLL / current program itself) loaded into the current process space. For this you can specify `CreateThread"module somefunction + 0x ##"`. This variant will generate a suspended thread that points to the specified function, if the specified function cannot be `GetProcAddress` obtained this is because Beacon will use to `SetThreadContextupdate` and will use this new thread to run the injected code, which is also a self-injection method that can provide you with a more favorable foothold.

SetThreadContext

Next is `SetThreadContext`, which is used in post-exploitation. One of the main thread method interim process tasks generated. The Beacon is `SetThreadContext` suitable for x86 -> x86, x64 -> x64 and x64-> x86. If you choose to use it `SetThreadContext`, place it in `CreateThread` after the option in the execution block. `SetThreadContext` when used; your thread will have a starting address that reflects the original execution entry point of the temporary process which is very nice.

NtQueueApcThread-s

Another way to suspend a process is to use it `NtQueueApcThread-s`. This method uses `NtQueueApcThread` which is a one-time function to queue up when the target thread wakes up next time. In this case, the target thread is the main thread of the temporary process. The next step is to call `ResumeThread`, this function wakes up the main thread of our suspended process, because the process has been suspended at this time, we do not have to worry about returning this main thread to the process. This method only applies to x86 -> x86 and x64 -> x64.

Determining whether to use `SetThreadContext` or `NtQueueApcThread-s` depends on you. In most cases I think the latter is obviously more convenient.

NtQueueApcThread

Another approach is through `NtQueueApcThread` it is like `NtQueueApcThread-sun` but it targets existing remote processes. This method needs to push the RWX stub to the remote process. This stub contains the code related to the injection. To execute the stub, you need to add the stub to the APC queue of each thread in the remote process. The stub code will be executed.

So what is the role of stubs?

First, the stub checks whether it is already running, and if it is, it executes nothing, preventing the injected code from running multiple times.

Then the stub will be called with the code and its parameters we injected `CreateThread`. This is done to let APC return quickly and let the original thread continue to work.

No thread will wake up and execute our stub. Beacon will wait about 200ms to start and check the stub to determine whether the code is still running. If not, update the stub and mark the injection as already running, and continue to the next item. This is `NtQueueApcThread` the implementation details of the technology.

At present, I have used this method a few times, because some security products have very little attention to this incident. In other words, OPSEC has paid attention to it, and it is indeed a memory indicator that promotes RWX stubs. It will also call the code of the remote process that we push `CreateThread`. The starting address of the thread does not support the module on the disk. Use `Get-InjectedThread` scan not effectively. If you think this injection method is valuable, please continue to use it. Pay attention to weighing its pros and cons. It is worth mentioning that this method is limited to x86 -> x86 and x64 -> x64.

CreateRemoteThread

Another way is via `CreateRemoteThread` which can be used literally as a remote injection technology. Starting with Windows Vista, injecting code across session boundaries will fail. In Cobalt Strike, `vanilla CreateRemoteThreadcovers` x86 -> x86, x64 -> x64 and x64 -> x86. The movement of this technology is also obvious. When this method is used to create a thread in another process, it will trigger event 8 of the system monitoring tool Sysmon. Such, Beacon has indeed implemented a `CreateRemoteThread` variant that `"module function + 0x ##"` accepts a

pseudo start address in the form `CreateThreadSimilarly`, Beacon will create its thread in the suspended state and use `SetThreadContext / ResumeThread` enable to execute our code. This variant is limited to x86 -> x86 and x64 -> x64. If the `GetProcAddress` specified function cannot be used, this variant will also fail.

RtlCreateUserThread

The last way Cobalt Strike executes blocks is `RtlCreateUserThread`. This way `CreateRemoteThread` functions is very enjoyable but has some limitations, it is not perfect and has flaws.

`RtlCreateUserThread` code will be injected across the session boundary. It is said that there will be many problems during the injection on Windows XP. This method will also trigger event 8 of the system monitoring tool Sysmon. One benefit is that it covers x86-> x86, x64-> x64, x64-> x86, and x86-> x64, the last case is very important.

x86-> x64 injection are in x86 Beacon session carried out. And for your *post-exploitation* generation process x64 tasks, `hashdump`, `mimikatz`, `execute-assembly` and `powerpick` modules are silent as x64. In order to achieve x86-> x64 injection, this method converts the x86 process to x64 mode and injects RWX stubs to facilitate calling from x64 `RtlCreateUserThread`. This technique comes from Meterpreter. RWX stubs are a pretty good memory indicator. I have long suggested: "Let the process stay in x64 mode as much as possible", the above situation is why I would say this, and it is also recommended to put one in all. So `process-inject` is the lowest way to have it, you can use it when there is no other `work execute blockRtlCreateUserThread`

0x04 How to live without process injection

When I was thinking about how to use these attack techniques flexibly, I was also thinking what to do if none of these methods work?

Process injection is a technique to `transfer payload / capability to migrate` to different processes (such as from desktop session 0 to desktop session 1), you can use the `runu` command to transfer to different processes without process injection, and you can specify the bot program as you To run child processes of any process. This is a way to introduce a session to another desktop session without process injection.

Process injection is also one of the methods to execute code without landing files on the target. Many *post-exploitation* functions in Cobalt Strike can choose to attack specific processes. Specifying the current Beacon process can use them without remote injection. This is self-injection.

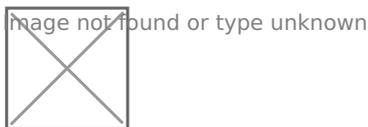
Of course, it is not perfect to execute code without a file on the ground. Sometimes it is best to put something on the disk. I have successfully compiled the keylogger tool into a DLL and put `c:\windows\linkinfo.dllit` into the `explorer.exe` process... We open sharing on the same system to share regularly captured key records, which helps me and my colleagues to operate under a highly censored situation. In this case, it is difficult to let the payload survive in the memory for a long time.

References:

- <https://blog.cobaltstrike.com/2019/08/21/cobalt-strikes-process-injection-the-details/>
- <https://blog.cobaltstrike.com/2016/11/03/agentless-post-exploitation/>
- <https://blog.cobaltstrike.com/2018/04/23/fighting-the-toolset/>
- <https://www.cobaltstrike.com/help-resource-kit>
- <https://www.cobaltstrike.com/help-artifact-kit>
- <https://www.youtube.com/watch?v=QvQerXsPSvc>

By Boschko

- My Hack The Box: <https://www.hackthebox.eu/home/users/profile/37879>
- My Website: <https://olivierlaflamme.github.io/>
- My GitHub: <https://github.com/OlivierLaflamme>
- My WeChat QR below:



Revision #3

Created 7 May 2020 16:10:12 by Boschko

Updated 2 November 2021 22:28:42 by Boschko