1. Introduction

Navigation

What is Aggressor Script?

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

History

Aggressor Script is the spiritual successor to Cortana, the open source scripting engine in Armitage. Cortana was made possible by a contract through DARPA's Cyber Fast Track program. Cortana allows its users to extend Armitage and control the Metasploit Framework and its features through Armitage's team server. Cobalt Strike 3.0 is a ground-up rewrite of Cobalt Strike without Armitage as a foundation. This change afforded an opportunity to revisit Cobalt Strike's scripting and build something around Cobalt Strike's features. The result of this work is Aggressor Script.

Aggressor Script is a scripting language for red team operations and adversary simulations inspired by scriptable IRC clients and bots. Its purpose is two-fold. You may create long running bots that simulate virtual red team members, hacking side-by-side with you. You may also use it to extend and modify the Cobalt Strike client to your needs.

Status

Aggressor Script is part of Cobalt Strike 3.0's foundation. Most popup menus and the presentation of events in Cobalt Strike 3.0 are managed by the Aggressor Script engine. That said, Aggressor Script is still in its infancy. Strategic Cyber LLC has yet to build APIs for most of Cobalt Strike's features. Expect to see Aggressor Script evolve over time. This documentation is also a work in progress.

How to Load Scripts

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

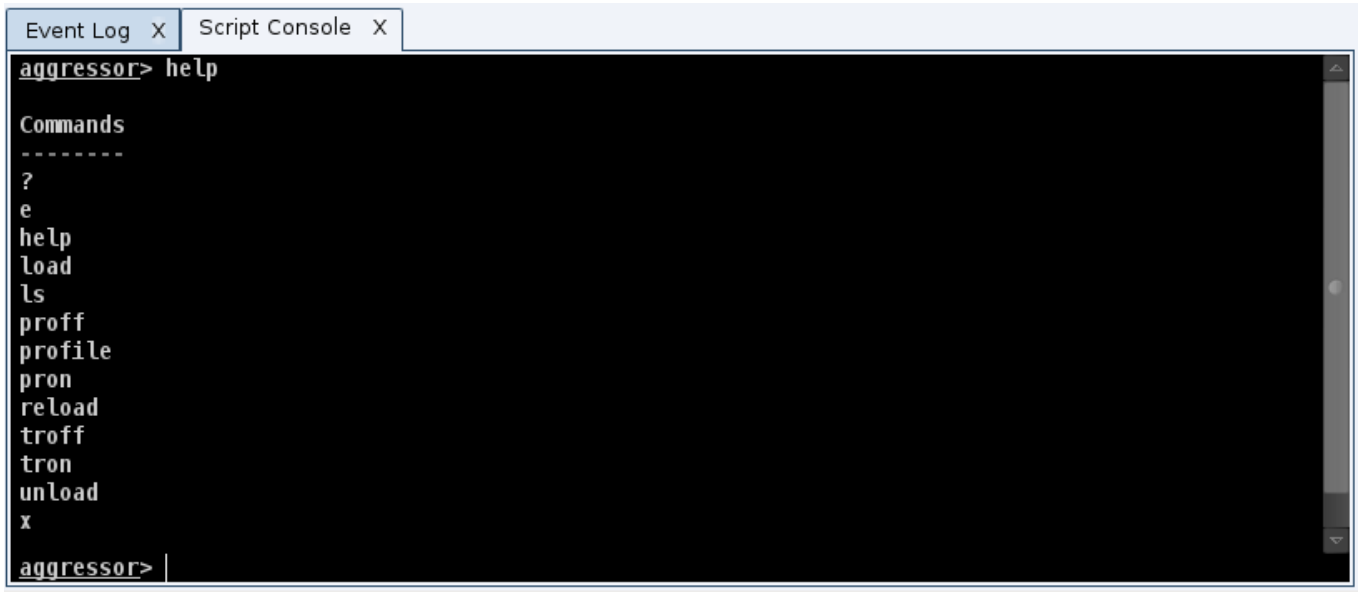
Cobalt Strike Script Loader

The Script Console

Cobalt Strike provides a console to control and interact with your scripts. Through the console you may trace, profile, debug, and manage your scripts. The Aggressor Script console is available via **View -> Script Console**.

The following commands are available in the console:

Command	Arguments	What it does
?	"*foo*" iswm "foobar"	evaluate a sleep predicate and print result
e	println("foo");	evaluate a sleep statement
help		list all of the commands available
load	/path/to/script.cna	load an Aggressor Script script
ls		list all of the scripts loaded
proff	script.cna	disable the Sleep profiler for the script
profile	script.cna	dumps performance statistics for the script.
pron	script.cna	enables the Sleep profiler for the script
reload	script.cna	reloads the script
troff	script.cna	disable function trace for the script
tron	script.cna	enable function trace for the script
unload	script.cna	unload the script
x	2 + 2	evaluate a sleep expression and print result



Interacting with the script console

Headless Cobalt Strike

You may use Aggressor Scripts without the Cobalt Strike GUI. The **agscript** program (included with the Cobalt Strike Linux package) runs the headless Cobalt Strike client. The agscript program requires four arguments:

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

These arguments connect the headless Cobalt Strike client to the team server you specify. The headless Cobalt Strike client presents the Aggressor Script console.

You may use agscript to immediately connect to a team server and run a script of your choosing. Use:

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

This command will connect the headless Cobalt Strike client to a team server, load your script, and run it.

A Quick Sleep Introduction

Aggressor Script builds on Raphael Mudge's Sleep Scripting Language. The Sleep manual is at:

http://sleep.dashnine.org/manual (http://sleep.dashnine.org/manual)

Aggressor Script will do anything that Sleep does. Here are a few things you should know to help keep your sanity.

Sleep's syntax, operators, and idioms are similar to the Perl scripting language. There is one major difference that catches new programmers. Sleep requires whitespace between operators and their terms. The following code is not valid:

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

This statement is valid though:

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

Sleep variables are called scalars and scalars hold strings, numbers in various formats, Java object references, functions, arrays, and dictionaries. Here are several assignments 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 with the @ and % functions. Arrays and dictionaries may reference other arrays and dictionaries. Arrays and dictionaries may even reference themselves.

Comments begin with a # and go until the end of the line.

Sleep interpolates double-quoted strings. This means that any white-space separated token beginning with a $ sign is replaced with its value. The special variable $+ concatenates an interpolated string with another value.

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

This will print out:

```
$a is: %(d => 4, b => 'bat', c => 'awesome language', a => 'apple') and
$x joined with $y is: Hello World3
```

There's a function called &warn. It works like &p (rfunctions.html#p)rintln, except it includes the current script name and a line number too. This is a great function to debug code with.

Sleep functions are declared with the sub keyword. Arguments to functions are labeled $1, $2, all the way up to $n. Functions will accept any number of arguments. The variable @_ is an array containing all of the arguments too. Changes to $1, $2, etc. will alter the contents of @_.

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

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

This script prints out:

```
58.0
```

In Sleep, a function is a first-class type like any other object. Here are a few things that you may see:

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

The $addf variable now references the &addTwoValues function. To call a function enclosed 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're interested in learning more about this. The following statements are equivalent and they do the same thing:

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

Sleep has three variable scopes: global, closure-specific, and local. The Sleep manual covers this in more detail. If you see local('$x $y $z') in an example, it means that $x, $y, and $z are local to the current function and their values will disappear when the function returns. Sleep uses lexical scoping for its variables.

Sleep has all of the other basic constructs you'd expect in a scripting language. You should read the manual to learn more about it.

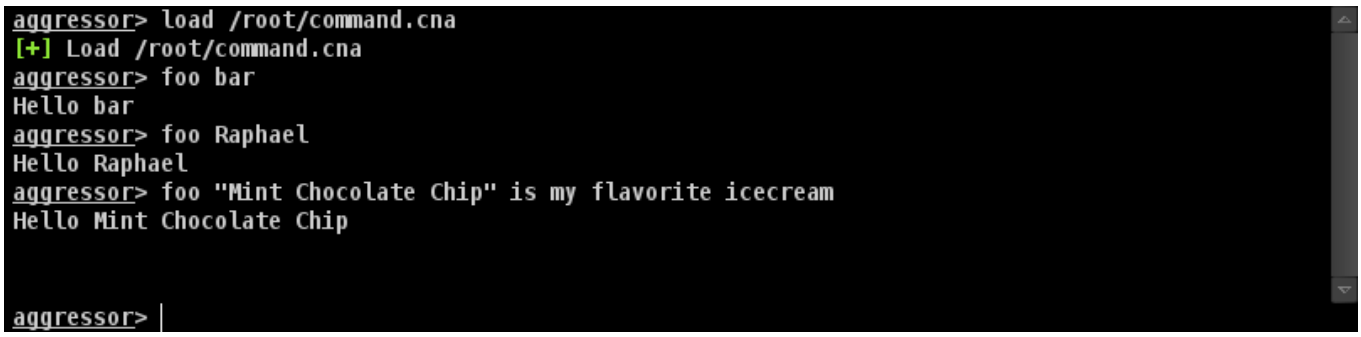
Interacting with the user

Aggressor Script displays output using Sleep's &p (rfunctions.html#p)rintln, &p (rfunctions.html#p)rintAll, &writeb, and &warn functions. These functions display output to the script console.

Scripts may register commands as well. These commands allow scripts to receive a trigger from the user through the console. Use the **command** keyword to register a command:

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

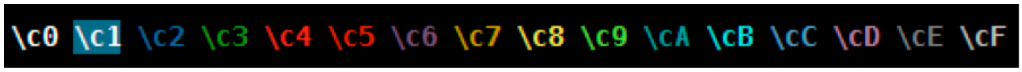
This code snippet registers the command foo. The script console automatically parses the arguments to a command and splits them by whitespace into tokens for you. $1 is the first token, $2 is the second token, and so on. Typically, tokens are separated by spaces but users may use "double quotes" to create a token with spaces. If this parsing is disruptive to what you'd like to do with the input, use $0 to access the raw text passed to the command.



Command Output

Colors

You may add color and styles to text that is output in Cobalt Strike's consoles. The `\c`, `\U`, nad `\o` escapes tell Cobalt Strile how to format text. These escapes are parsed inside of double-quoted strings only.



Color Options

The `\cX` escape colors the text that comes after it. X specifies the color. Your color choices are:

The `\U` escape underlines the text that comes after it. A second `\U` stops the underline format.

The `\o` escape resets the format of the text that comes after it. A newline resets text formatting as well.