ShellPcapFication (SPF)

A Shell Framework

Documentation and Language Specification

Mohamad Fadel Mokbel
http://www.mfmokbel.com
mfmokbel [AT] live [DOT] com
Twitter: @MFMokbel

SPF (**S**hell**P**cap**F**ication), is a shell framework that provides a sophisticated abstraction layer for TShark[1] (console-based version of Wireshark) and Windows command shell interpreter. It features a custom, unique and simple declarative language called **Eros**[2] that consists of only two constructs, four keywords, three Input operators, auxiliary logic, a function call operator, an INSERT statement, a specifier, and an *include* preprocessing directive, among others. Additionally, a set of built-in helper commands are also provided by SPF to simplify interaction with **Eros** in a dynamic way.

SPF main features include:

- The democratization of writing and sharing a standardized set of constructs based on **Eros** language
- The capability to use different constructs as building blocks to form complex operations
- Simplification of repetitive tasks
- Rich shell functionality
- Automation of Exploit Kit detection
- Protocol specific features/fields extraction
- Building self-contained and easy to manage self-explanatory units/constructs
- Functioning as a signature detection system (based on TShark powerful protocol dissectors)

[1] https://www.wireshark.org/docs/man-pages/tshark.html
[2] In reference to the Greek God of love, procreation, and sexual desire

[Updated on November 05, 2017]

Table of Contents

Table of Tables

Table of Figures

INTRODUCTION

SPF is a shell framework (console based) that provides seamless interaction with TShark and Windows command shell interpreter (CMD) via a housekeeping custom developed declarative language called **Eros**. Interaction with TShark and CMD is facilitated through two unique distinct constructs known as SPF() and WIN(), respectively. SPF() construct LOGIC uses the syntax of read/display filters. WIN() construct LOGIC uses whatever CMD runs (CMD commands, PowerShell, WMI, Batch, etc).

As shown in Figure 1, these are the constructs skeleton that SPF (SPF is the name of the framework and one of the supported constructs, when the acronym SPF is mentioned alone, it refers to the framework unless noted otherwise) supports. Each construct is given a unique and descriptive name under the **NAME** keyword that will be made accessible from the shell as an execution unit. The **LOGIC** keyword is where the syntax of a construct is defined. For SPF(), it takes the syntax of TShark read/display filters, thus, it is a prerequisite that you understand how TShark works and how to write syntactically and semantically correct display filters against any of the supported protocols. For more information on display filters, please refer to "*Wireshark filter syntax and reference*"[3] and "*DisplayFilters*"[4]. For WIN() construct, the syntax is as stated earlier. SPF() and WIN() constructs constitute the knowledge base of SPF.

```
SPF() -> hide
{
    NAME  = <cmd name>
    LOGIC = <logic definition/syntax>
    INFO  = <cmd info/help>
    TAG   = <cmd tag/metadata>
}
```

```
WIN() -> hide
{
    NAME  = <cmd name>
    LOGIC = <logic definition/syntax>
    INFO  = <cmd info/help>
    TAG   = <cmd tag/metadata>
}
```

Figure 1 SPF Constructs

The **INFO** keyword is where you give a brief description of the construct. This is also accessible from SPF shell. The **TAG** keyword allows you to non-hierarchically assign multiple keywords/metadata to the construct. The **TAG** keyword is helpful when you want to search for a set of constructs that belongs to you, belongs to a specific protocol, etc. This is also accessible from SPF shell.

CONSTRUCTS SPECIFICATION (LEXICAL CONVENTIONS)

This section provides a detailed description of each of the constructs declarations and definitions in terms of language specifications.

List of allowed keywords/tokens: **NAME**, **LOGIC**, **INFO**, and **TAG**

- Keywords declaration rules
    - Case sensitive, uppercase only

[3] https://www.wireshark.org/docs/man-pages/wireshark-filter.html
[4] https://wiki.wireshark.org/DisplayFilters

- Positioned anywhere on the line
- Tabs and whitespaces are allowed before and after each of the keywords (before the assignment operator **=**)
- Each keyword starts on a new line and ends with the assignment operator **=**

- Keywords definition rules:

  - The following definitions take action after the assignment operator (in-order)

    - **NAME**
      - Whitespaces are optional (right after the assignment operator **=**)
      - List of allowed characters [a-zA-Z0-9_]{1,64}
    - **LOGIC**
      - Only first whitespace (right after the assignment operator **=**) is optional. Every other whitespace is accounted for
      - It takes anything for a minimum of one character
    - **INFO**
      - Only first whitespace (right after the assignment operator **=**) is optional. Every other whitespace is accounted for
      - It takes anything for a minimum of one character
    - **TAG**
      - Only first whitespace (right after the assignment operator **=**) is optional
      - List of tags/metadata, all separated by the operator **|**
      - List of allowed characters [a-zA-Z0-9_|]+
      - This keyword is optional, and could be omitted in a given construct

    - All keywords definitions cannot span more than one line

- Constructs declaration rules (in-order)

  - The name of the constructs **SPF()** or **WIN()** could be positioned anywhere on a new line. It could be preceded with zero or more number of tabs and whitespaces, and followed by zero or more number of tabs, whitespaces or newlines.
  - The specifier "**-> hide**" after the construct name is optional
    - Zero or more number of spaces and tabs is/are allowed between "**->**" and "**hide**"
    - The string "**hide**" is case sensitive (lowercase only)
  - The name of the construct is followed by the opening delimiter **{**
    - The opening delimiter could be followed by zero or more number of tabs, spaces or newlines.
  - The end of the construct is indicated by the closing delimiter **}**

- Constructs definition rules (in-order)

- Each construct's definition has to include the keywords: **NAME**, **LOGIC**, **INFO** (in the shown order) for the construct to be valid and consumed by SPF
- The keyword **TAG** is optional
    - Should it be included, it has to be the last keyword
- There could be any number of newlines between the opening delimiter **{**, keywords, and closing delimiter **}**
- C++ line based comments **//**<comment> are valid anywhere in the body of the construct
- Auxiliary Logic definitions (discussed later in the documentation under "Global Auxiliary Logic Definitions") are allowed at the beginning of the construct only, and although they are defined inside a construct, their scope and accessibility are global nonetheless

SPECIFIER "-> hide"

The specifier "**-> hide**" is optional. It is used to indicate that a given construct is not visible through the Helper CMD ***getallcmds*** (please refer to the SPF HELPER COMMANDS section for more information). However, that does not mean that you cannot use the construct from the shell or through other constructs. This specifier helps in cases where a construct is not meant to be used as a standalone executable command.

CMD.spf SPECIFICATION

- It is the master SPF CMD file that contains the list of all **SPF()** and **WIN()** unique constructs
- File name is case sensitive and hardcoded. It cannot be changed
- The directive **<SPF CONTROL COMMANDS>** has to be present anywhere in the file, otherwise SPF parser will bail out parsing the rest of the file's content

include PREPROCESSING DIRECTIVE SPECIFICATION

#include <SPF CMD filename>

The ***#include*** preprocessing directive allows the inclusion of other files that contain **SPF()** or **WIN()** constructs. The structure of those files is similar to CMD.spf and should have the directive **<SPF CONTROL COMMANDS>** inserted in every referenced file. Nested includes are allowed. There is no restriction on the included file filename, however, it is recommended that you give it a descriptive name and keep the extension "spf".

The syntax of this directive is as follows:

- Zero or more number of whitespaces and tabs are allowed before the ***#include*** directive string
- Zero or more number of whitespaces and tabs are allowed after the ***#include*** directive string
- Referenced ***SPF CMD filename*** between **<** and **>** cannot be empty
- No characters are allowed after the end of the directive definition (after **>**)
- Each directive has to be present on a new line

Note: There is no limitation on the number of constructs in a given file.

INPUT OPERATORS

The power of **Eros** language lies in its capability to parametrize the syntax defined in the **LOGIC** keyword of a given construct using a set of preprogrammed unique Input Operators.

Table 1 shows all supported Input operators that are used in the definition of the **LOGIC** keyword.

Input operators	Description
[%_ARG_%]	Takes input string from the command line. No whitespaces are allowed. No restriction on the number of times this operator is referenced in a given **LOGIC** keyword definition.
[%_LIST<<filename>>_%]	Takes the content of a text file. Each required entry in the text file has to be put on a newline. This input operator can be generalized with the **[%_ARG_%]** operator such that reference to the filename is entered as a command line argument from SPF shell as follows: **[%_LIST<<[%_ARG_%]>>_%]** Should a filename contain backslashes, then they have to be escaped. When invoking a complete SPF shell command, there can exist only one instance of this operator, otherwise, SPF CMD parser will spit an error message and not consume referenced construct. Should this operator be present in a complete SPF shell command, then SPF will execute said command in an iterative manner such that on every execution it reads one line at a time as an input argument to the command. Thus, input from a LIST file cannot contain any SPF commands as it won't be parsed nor evaluated. This is a design decision. An entry in the referenced file could be given the **PRINT** operator **[%_PRINT<<MSG>>_%]<DATA>** Above operator gives each line a message to print to the console before "executing" the **<DATA>** part (note there is no whitespace between the **PRINT** operator and the **<DATA>** part). This operator is optional. The rules for this operator are as follow: • It has to start on a new line • A zero or more number of whitespaces and tabs are allowed at the beginning of the line • Followed by the optional **PRINT** operator **[%_PRINT<<MSG>>_%]**, o The <MSG> can take any characters except for \r, \n and %. o For example, **[%_PRINT<Angler Exploit Kit URI Pattern>_%]**

Table 1 List of Input Operators

CALL OPERATOR

The function call operator allows you to call other constructs **LOGIC** implementation as building blocks to build sophisticated constructs. Call operator is used in the definition of the **LOGIC** keyword. It has the following syntax:

[CALL(<SPF/WIN CMD>)]

There is no limitation on the number of times this operator is used. SPF first will validate referenced SPF/WIN command(s) in every used **CALL** operator, and if the command is invalid, no further parsing or unpacking of the command(s) is performed until the offending entry is fixed. Moreover, should you end up in a situation where a self-referenced command is encountered, SPF will report it, and processing of the **CALL** operator is halted until offending entry is fixed.

Note: An SPF() construct **LOGIC** can make calls to a WIN() construct LOGIC, but not vice versa. This is because depending on the sequence and order of SPF/WIN commands used, different execution units take precedence. Nonetheless, it is possible to make calls from a WIN() construct **LOGIC** to an SPF() construct **LOGIC** by referencing TShark process directly.

GLOBAL AUXILIARY LOGIC DEFINITIONS

Auxiliary Logic (AL) definitions are global (with universal lexical scope) non-executable named statements that are used with SPF/WIN constructs as building blocks. Below is the skeleton of this AL definition:

```
L.<auxiliary logic definition name> = <logic definition>
```

Any SPF/WIN construct **LOGIC** definition can reference an AL definition via the operator statement

[INSERT(<auxiliary logic definition name>)]

Thus, whenever SPF parser finds an instance of the **INSERT** operator statement in a given construct's **LOGIC** definition, it automatically unpacks it with the referenced AL definition implementation.

AL definitions provide an easy way to expose/share part of a LOGIC definition across multiple constructs. As stated, AL definitions are not executable statements on their own nor are they considered commands. They are meant to be used in cases where a shared LOGIC sub-statement is needed by different constructs, but the shared sub-statement does not qualify for a standalone construct. Therefore, AL definitions do not take any of the other keywords that SPF/WIN constructs take.

Note that AL definitions can be written anywhere except inside an SPF/WIN construct. Additionally, they cannot be used as SPF shell commands.

AL declarations and definitions in terms of language specifications include:

- Keywords declaration rules
    - It starts with **L.**
        - Case sensitive, uppercase only

- Positioned anywhere on the line
- After the dot, it is followed by the auxiliary logic definition name, which takes the following list of allowed characters [a-zA-Z0-9_]{1,64}
- Tabs and whitespaces are allowed before and after each of the keywords (before the assignment operator =)
- Each keyword starts on a new line and ends with the assignment operator =

- Keywords definition rules:

  - The following definitions take action after the assignment operator (in-order)

    - Only first whitespace (right after the assignment operator =) is optional. Every other whitespace is accounted for
    - It takes anything for a minimum of one character
    - The reference statement operator **[INSERT(<AL definition name>)]** is not allowed to be used with any AL definition statements (this is a design decision)

The following example demonstrates how AL definitions work.

First, we define a new AL as follows:

```
L.pln = findstr/n ^^
```

Above AL with the name "pln" is responsible for printing line numbers.

Second, let's take the following LOGIC implementation for an SPF() construct with the name "**domains**" and reference above AL definition in it using the **INSERT** statement:

```
SPF()
        {
        NAME  = domains
        LOGIC = -Y "dns && dns.flags.response eq 0" -T fields -e dns.qry.name | [INSERT(pln)]
        INFO  = Get all DNS queries
        TAG   = mokbel|dns
        }
```

When SPF parser encounters a valid reference to an AL definition via the INSERT statement, it will supplants it with its implementation as follows:

```
        LOGIC = -Y "dns && dns.flags.response eq 0" -T fields -e dns.qry.name | findstr/n ^^
```

IMPLICIT CONSTRUCTORS

Implicit Constructors (IC) are special functions that can be used to initialize the logic, of all constructs at once, a set of constructs, or a construct, by either a prefix or a suffix or both, via Auxiliary Logic definition(s).

ICs provide a powerful mechanism for default initialization of SPF() or WIN () constructs logic. Each construct type, that is SPF() or WIN(), has its own Prefix and Suffix Implicit Constructors specifiers. It makes reading the LOGIC's definition cleaner and void of non-logic related arguments.

"Implicit" means that no reference is required anywhere in the construct. ICs are similar in spirit of what a constructor is like in a modern high level language.

Prefix ICs are used for prefixing a given construct's logic definition (SPF() or WIN()) with whatever default value as referenced by the AL definition.

Suffix ICs are used for appending a given construct's logic definition (SPF() or WIN()) with whatever default value as referenced by the AL definition.

ICs have two scoping boundaries, Global and Local.

- Global is for prefixing or appending ("suffixing") all SPF() or WIN() constructs at once.
- Local is for prefixing or appending ("suffixing") a set of SPF() or WIN() constructs, or a construct.

❖ Global SPF and WIN Implicit Constructors skeletons:

```
Prefix.SPF = {*} -> [L.<auxiliary logic>]

Suffix.SPF = {*} -> [L.<auxiliary logic>]

Prefix.WIN = {*} -> [L.<auxiliary logic>]

Suffix.WIN = {*} -> [L.<auxiliary logic>]
```

- Keywords declaration rules

  - **Prefix.SPF**, **Suffix.SPF**, **Prefix.WIN** and **Suffix.WIN**
    - Case sensitive
    - Positioned anywhere on the line
    - Tabs and whitespaces are allowed before and after each of the keywords (before the assignment operator **=**)
    - Each keyword starts on a new line and ends with the assignment operator **=**

- Keywords definition rules (take action after the assignment operator)

  - The special operator {*****} is used to indicate that it is a Global IC, and it is always followed by the specifier -> pointing to an existing AL definition L.<AL definition name>
    - Tabs and whitespaces are allowed before and after each of the operators (after the assignment operator **=**

- There can be only one Global IC per construct type, the parser will complain otherwise printing to the console a message similar to "*Only one Global <Prefix|Suffix> <WIN|SPF> Implicit Constructor is allowed. Only the first will be parsed.*"

- If more than one Global IC per construct type is present, then, only the first will be taken and the rest are discarded

For example, let's say we have the following AL definition (note the space at the end as follows " -Y "):

```
L.display_filter = -Y
```

To define a Global SPF() IC, we write: **Prefix.SPF** = {*} -> [L.display_filter]

Above Global SPF() IC will prefix every valid SPF() construct with the definition of the "display_filter" AL, that is " -Y ".

For excluding a given construct from being initialized with respect to a Global IC, the following IC specifier could be used:

Prefix.SPF = {!*<construct name>,…, !<construct name>***}**

Suffix.SPF = {!*<construct name>,…, !<construct name>***}**

Prefix.WIN = {!*<construct name>,…, !<construct name>***}**

Suffix.WIN = {!*<construct name>,…, !<construct name>***}**

Keywords declaration and definition rules are the same of Local SPF and WIN Implicit Constructors skeletons (please read below). Note the absence of a reference to an AL definition in the skeletons.

The exclusion specifier is only relevant when a respective Global IC is present.

❖ Local SPF and WIN Implicit Constructors skeletons:

Prefix.SPF = {*<construct name>,…, <construct name>***} -> [L.**<auxiliary logic>**]**

Suffix.SPF = {*<construct name>,…, <construct name>***} -> [L.**<auxiliary logic>**]**

Prefix.WIN = {*<construct name>,…, <construct name>***} -> [L.**<auxiliary logic>**]**

Suffix.WIN = {*<construct name>,…, <construct name>***} -> [L.**<auxiliary logic>**]**

- Keywords declaration rules: The same as in the case of the Global ICs
- Keywords definition rules (take action after the assignment operator)

- If more than one construct is referenced, then each is separated by a comma, and each could be placed on a new line
- A minimum of one valid SPF() or WIN() construct is required
- There can be multiple definitions of each of the Local SPF() and WIN() ICs, whereby each referencing different construct(s). The same construct cannot be referenced more than once, even if it is pointing to different AL definitions, the parser will complain otherwise printing to the console a message similar to "*Referenced construct name "<construct name>" in the Local <SPF/WIN> <Prefix/Suffix> Implicit Constructor is already assigned an AL value. Only the first instance is taken.*"

- It is not allowed to reference a WIN() construct inside an SPF IC, the parser will complain otherwise printing to the console a message similar to "*Referenced construct name "<WIN construct name>" in the Local SPF <Prefix/Suffix> Implicit Constructor is not an SPF construct. Please rectify offending command and try again.*"
- It is not allowed to reference an SPF() construct inside a WIN IC, the parser will complain otherwise printing to the console a message similar to "*Referenced construct name "<SPF construct name>" in the Local WIN <Prefix/Suffix> Implicit Constructor is not a WIN construct. Please rectify offending command and try again.*"
-

To define a Local SPF IC, we write:

```
Prefix.SPF = {geturi, domains} -> [L.display_filter]

Prefix.SPF = {alldns} -> [L.display_filter]
```

Each of the IC's definitions can be inspected from the shell via the Helper cmd **gic** (please refer to the SPF HELPER COMMANDS section for more information).

What happens in case of both a Global IC and a Local IC are defined for the same construct's type and specifier? Those defined in the related Local IC will be always parsed exclusively, irrespective of whether both, the Global and Local IC share the same AL definition.

MULTI COMMAND UNIT (MCU)

A MCU provides a mechanism to automate the execution of different SPF() and WIN() constructs in addition to any other Windows based shell commands. Figure 2 shows the skeleton of the MCU. There is no limit on the number of MCUs present in an SPF file. Moreover, they are standalone self-contained units that do not interact with each other.
MCU declaration and definition in terms of language specifications include:

- MCU declaration rules

  - Writing a MCU requires the usage of the keyword **MCU(<unit name>)**. And, the MCU is given a name between the parentheses. **MCU(<unit name>)** could be positioned anywhere on a new line. It could be preceded with zero or more number of tabs and whitespaces, and followed by one or more number of tabs, whitespaces or newlines.
    - A **MCU** unit name takes the following list of allowed characters [a-zA-Z0-9_]{1,64}

- The declaration of a MCU is followed by the opening delimiter **{**
    - The opening delimiter could be followed by one or more number of tabs, spaces or newlines.
- The end of the MCU is indicated by the closing delimiter **};**

- MCU definition rules

    - A MCU is allowed to be empty
    - Reference to any of the defined SPF() and WIN() constructs in addition to any Windows based shell commands has to start on a new line
        - All leading whitespaces and tabs are trimmed and not consumed
    - There could be any number of newlines between the opening delimiter **{**, commands, and closing delimiter **};**
        - Empty lines are not consumed nor processed
    - C++ line based comments **//** <comment> are valid anywhere in the body of the MCU

```
MCU(<unit name>)
{
    <SPF/WIN or Win CMD commands>
    …
    …
    <SPF/WIN or Win CMD commands>
};
```

Figure 2 Multi Command Unit (MCU) Skeleton

A defined MCU commands are not verified until the MCU is invoked. Moreover, the execution order is top-to-bottom order, line by line. Interaction with MCU from the SPF shell is done via the helper CMDs ***exmcu*** (to execute a given MCU) and ***getmculist*** (to get a list of all available MCUs).

The following example demonstrates how MCU definition works:

```
MCU(test)
        {
        // This is just for testing
        echo [ Executing geturi SPF() CMD ]
        geturi GET

        echo [ Executing arch WIN() CMD ]
        arch
        };
```

The MCU with the name "**test**" is responsible for printing/echoing messages to the console window and referencing an already defined constructs, **geturi** ***GET*** and **arch**. Executing this MCU from the SPF shell is done via the Helper CMD:

exmcu **test**

CONFIGURATION FILE

The configuration file is responsible for configuring some of SPF's path dependencies, among other options.

- Configuration file name is "**SPF.cfg**", and cannot be changed. It has to be present in the same folder of SPF executable
- Upon executing SPF process, "**SPF.cfg**" is the first to be read and parsed. Some of those configuration options will influence how SPF will interact with the user's environment
- The directive **<SPF CONFIGURATION FILE>** has to be present anywhere in the file, otherwise SPF parser will bail out parsing the rest of the file's content
- The strings **[PATHS TO SET]**, **[OPTIONS TO SET]**, among other similar strings are only for documentation purposes and aren't checked for by SPF

- Every configuration option has to end with "**;**" (semicolon)
- Every configuration option has to be on a new line
- The declaration of every configuration option takes the following format pattern
    - <zero or more whitespaces><configuration option>< zero or more whitespaces> =
- The definition of every configuration option takes the following format pattern after the assignment operator "**=**"
    - <zero or more whitespaces><configuration option value (it can't be empty)>**;**

Configuration Options
SPF_CMD_FILE_PATH = <>;
This option sets the path to the "**CMD.spf**" file. Backslashes has to be escaped. If "**CMD.spf**" is in the same directory of SPF process, then this option has to be set with the value **LOCAL**. The value **LOCAL** has to be uppercase. For example, **SPF_CMD_FILE_PATH = LOCAL;** **Or** **SPF_CMD_FILE_PATH = C:\\Program Files\\SPF\\;**
TSHARK_EXE_PATH = <>;
This option sets the path to TShark executable. Backslashes has to be escaped.
PCAP_DIR_PATH = <>;
This option sets the path to a folder that contains "all" pcaps. Backslashes has to be escaped. This option value could be overwritten with the Helper CMD ***setpcappath***.
DEFAULT_PCAP_NAME = <>;
This option sets a default pcap filename. Value can be retrieved with the Helper CMD ***getpcap*** or set/changed in-memory with ***setpcap***.
LOAD_CMD_SPF_FILE = <>;
This option requests from SPF process whether to load and parse "**CMD.spf**" at runtime or not. This option takes either of the values: **TRUE** or **FALSE**. Values are uppercase.
HISTORY_DIR_PATH = <>;
This option sets the path to the "**.spf_history**" file. Backslashes has to be escaped. If "**.spf_history**" is in the same directory of SPF process, then this option has to be set with the value **LOCAL**. The value **LOCAL** has to be uppercase. For example, **HISTORY_DIR_PATH = LOCAL;** **Or** **HISTORY_DIR_PATH = History\\;**

This file is responsible for storing all previously executed commands.

LOAD_HISTORY_FILE = <>;

This option requests from SPF process whether to load and parse the content of "**.spf_history**" at runtime or not, and make all stored commands available from the shell through the Helper CMDs ***history*** and ***exh***. This option takes either of the values: **TRUE** or **FALSE**. Values are uppercase.

If this option is set to **TRUE** and the file "**.spf_history**" doesn't exist on disk, then SPF will create it automatically. Additionally, every executed SPF shell command is saved/pushed to the history file at the time of execution.

DEPENDENCIES

TShark and Windows command shell interpreter.

TShark binaries are shipped with the main installer of Wireshark. And, it is usually located under the same directory of main Wireshark installation folder under the name tshark.exe.

Windows command shell interpreter is Windows OS version dependent. No reference to the process is required.

SPF HELPER COMMANDS

The list of SPF Helper Commands in Table 2 functions as a conduit to interact with defined SPF CMDs and the shell itself in a dynamic way. It is recommended that you familiarize yourself first with each of the Helper CMDs to get a better understanding of the shell's capabilities as well as its limitations.

Helper CMD	Argument	Description
setpcap	*N/A*	set the name of the pcap you want to work with. To work with multiple pcaps at the same time, set the pcap name to "***AF***" (acronym for **A**ll**F**iles). This can also be set from the configuration file via the option **DEFAULT_PCAP_NAME**. Note that all directories and nested sub-directories will be parsed from the root directory which is set by the configuration option **PCAP_DIR_PATH** or through the Helper CMD ***setpcappath***.
getpcap	*N/A*	print the name of the currently set pcap
setpcappath	*<pathtopcap>*	set the path to the pcap you want to work with (in-memory only): ex., **setpcappath** *C:\Users\M.F\Desktop\SPF*
getpcappath	*N/A*	print the name of the currently set pcap path
loadcmdfile	*N/A*	(re)load the content of SPF CMDs stored in CMD.spf (in case of an external update/change to the constructs)

<table>
<tr><td>getallcmds</td><td>N/A</td><td>Print a list of all available cmds in CMD.spf, internal, and helper cmds, among others. If an SPF construct is given the specifier “-> hide”, then said cmd won’t show up in the list</td></tr>
<tr><td>getinclist</td><td>N/A</td><td>print the name(s) of all include files in CMD.spf</td></tr>
<tr><td>info</td><td><SPF/WIN CMD></td><td>print info/help for a given SPF/WIN cmd</td></tr>
<tr><td>logic</td><td><SPF/WIN CMD></td><td>print logic implementation for a given SPF/WIN cmd</td></tr>
<tr><td>auxl</td><td><list of AL names></td><td>print a list of all available Auxiliary Logic definitions if no argument(s) is given. Otherwise, you can print one or multiple specific ALs at once.
ex., auxl <AL name> <AL name> … <Al name></td></tr>
<tr><td>gic</td><td><sub-cmd></td><td>print the AL name, or the list of excluded constructs a given Implicit Constructor (IC) holds. This Helper CMD supports the following list of print sub-commands:

<table>
<tr><th>gic sub-cmd</th><th>Description</th></tr>
<tr><td colspan="2">Global Implicit Constructors</td></tr>
<tr><td>pre.spf.g</td><td>Global SPF Prefix IC</td></tr>
<tr><td>pre.win.g</td><td>Global WIN Prefix IC</td></tr>
<tr><td>suf.spf.g</td><td>Global SPF Suffix IC</td></tr>
<tr><td>suf.win.g</td><td>Global WIN Suffix IC</td></tr>
<tr><td colspan="2">Specific Implicit Constructors</td></tr>
<tr><td>pre.spf.s</td><td>Specific SPF Prefix IC</td></tr>
<tr><td>pre.win.s</td><td>Specific WIN Prefix IC</td></tr>
<tr><td>suf.spf.s</td><td>Specific SPF Suffix IC</td></tr>
<tr><td>suf.win.s</td><td>Specific WIN Suffix IC</td></tr>
<tr><td colspan="2">Exclusion Implicit Constructors</td></tr>
<tr><td>pre.spf.e</td><td>Exclusion SPF Prefix IC</td></tr>
<tr><td>pre.win.e</td><td>Exclusion WIN Prefix IC</td></tr>
<tr><td>suf.spf.e</td><td>Exclusion SPF Suffix IC</td></tr>
<tr><td>suf.win.e</td><td>Exclusion WIN Suffix IC</td></tr>
</table>

ex., gic pre.spf.s</td></tr>
<tr><td>ulogic</td><td><SPF/WIN CMD></td><td>update (in-memory only) the unpacked LOGIC implementation for a given SPF/WIN cmd. although it modifies the logic, no verification on the updated logic is performed. Caution should be exercised when using this cmd as it might break other interdependent cmds in the same session.
The default/current LOGIC implementation is copied to the clipboard for ease of modification, and cleared afterwards</td></tr>
<tr><td>tag</td><td><SPF/WIN CMD></td><td>print a list of tags/metadata for a given SPF/WIN cmd</td></tr>
<tr><td>gcwt</td><td><tag> <tag> … <tag></td><td>print a list of all available SPF/WIN CMDs that match entered list of tags. Matching CMD would have to match all tags</td></tr>
<tr><td>history</td><td>N/A</td><td>print a list of all previously executed commands</td></tr>
<tr><td>chistory</td><td>N/A</td><td>clear the history list of all previously executed commands</td></tr>
<tr><td>exh</td><td><list of history cmds></td><td>(re)execute list of previously executed commands</td></tr>
</table>

		ex., **exh** *1 3 5*
getmculist	*N/A*	print list of all available Multi Command Units
exmcu	*<MCU name>*	execute list of SPF/WIN commands defined in a Multi Command Unit
about	*N/A*	print SPF version and author information
version	*N/A*	same as **about**
helper	*N/A*	print this list!
exit	*N/A*	exit the shell

Table 2 List of SPF Helper Commands

ORDER OF EVALUATIONS

From Loading and Parsing, to Evaluation and Execution

It is important that you understand how SPF goes about the order of loading an SPF CMD file, parsing constructs and Input operators, to evaluation and execution steps.

Following are the steps SPF takes upon execution (in the order shown):

1. Reading and parsing of the configuration file "**SPF.cfg**"
2. Loading and parsing of auxiliary logic commands in "**CMD.spf**", and in all files used in the preprocessing ***#include*** directive
3. Loading and parsing of all SPF/WIN commands in "**CMD.spf**"
    a. Checking the LOGIC definition of every SPF/WIN CMD for the **[INSERT<auxiliary logic cmd>]** and unpack it
    b. This includes parsing of all **TAG**s keywords,
    c. Determining how many **[%_ARG_%]** input operator are there in a every SPF/WIN command,
    d. And checking if a given SPF/WIN command has the **[%_LIST<filename>_%]** input operator
4. Loading and parsing of all MCUs in "**CMD.spf**"
    a. Validation of SPF/WIN commands in a given MCU is deferred until time of invokation
5. Reading and parsing of all files used in the preprocessing ***#include*** directive (step 3 steps are exercised again)
6. Checking for **CALL** operators and unpacking them
7. Loading and parsing of all Implicit Constructors in "**CMD.spf**", and in all files used in the preprocessing ***#include*** directive
8. Depending on the flag setting of the configuration parameter LOAD_HISTORY_FILE, SPF will either load the "**.spf_history**" file at runtime, or not
9. Opening of SPF shell
10. Every entered command (first command) is first checked against any of the Helper CMDs, and if no match is found, then
    a. Command is checked against the dynamically resolved list of WIN commands, and if a match is found,

i. It is a given that every subsequent command shall belong to the list of WIN construct commands, otherwise behaviour is Windows command shell interpreter dependent

b. Then, execution is delegated to Windows command shell interpreter
c. Otherwise, command is checked against the dynamically resolved list of SPF commands, and if a match is found,
d. Every subsequent command on the line is validated accordingly taking into consideration the presence of any **[%_ARG_%]** Input operator
e. Otherwise, if the command at (c.) is not valid, then
f. Entered command is delegated to Windows command shell interpreter

11. If entered command is found in the list of Helper CMDs, then execution is delegated to a special execution unit
12. Before executing the complete SPF shell command, if an SPF/WIN command was found to contain a **LIST** Input operator, then execution is delegated to a special executor that can handle such operator.

RESERVED KEYWORDS

Reserved keywords are keywords that cannot be used as SPF/WIN command names. This is also dependent on the order of evaluation of each entered command as well as the sequence of commands.

None of SPF/WIN commands names can take any of the **Helper CMDs** nor any of Windows command interpreter predefined commands. This is because first entered SPF shell command goes through different execution units before determining which one to invoke. For more information, please refer to section "ORDER OF EVALUATIONS". Although it is possible to give an SPF/WIN construct the same name of a predefined **Helper CMD**, its validation is dependent on where in the construct it happens to be referenced/located as well as in the final SPF shell command, however, it is highly recommended not to do so.

INTERNALS

CONSTRUCTS COMMAND PROCESS EXECUTION

Internally, SPF has to build the full SPF shell command and pass it to either of TShark or Windows command shell interpreter processes to execute it.

For TShark, the full internal SPF shell command skeleton is as follows:

```
<Path To TShark> -r <pcap directory path><pcap filename> <SPF() construct(s) LOGIC>
```

Of importance to note is TShark option "**-r**". This option takes the name of the pcap you want to work with as input argument. This is set via the Helper CMD **setpcap**. TShark defines this option as follows:

"*Read packet data from infile, can be any supported capture file format (including gzipped files). It is possible to use named pipes or stdin (-) here but only with certain (not compressed) capture file formats (in particular: those that can be read without seeking backwards).*"

The rest of the options are self-explanatory.

For Windows command shell interpreter, everything in the final SPF shell command is passed to it as it is.

STRUCTURES

Internally, SPF consists of four main global objects whereby each is responsible for maintaining the following constructs:

- SPF External CMDs List
  - This object references all external SPF() constructs
- SPF Internal CMDs List
  - This object references all internal SPF() constructs (if any)
- WIN External CMDs List
  - This object references all external WIN() constructs
- WIN Internal CMDs List
  - This object references all internal WIN() constructs (if any)

Internal constructs are meant to represent complex functionalities written as part of the framework's code that are not possible to write via external constructs.

OTHERS

Note that SPF is only ASCII aware.

SPF is programmed in the C++ language using a mix of C++11 and C++14. Additionally, it is compiled using MS Visual Studio 2015. Compilation code optimization is set to full with Whole Program Optimization Enabled.

One of the primary motivations for working on this project was to explore the new features of C++11 and C++14.

The first character of the word-forming element "**F**ication" in the name of the framework SPF (**S**hell**P**cap**F**ication) is capitalized for no particular reason other than the fact that I want it a minimum of three characters acronym name.

SPF error reporting for parsing of invalid constructs and AL definitions is limited. If a construct or AL ends up in the list of all SPF commands, it means it passed all validations and checks, otherwise, it is not valid and requires correction.

COLLABORATION

Since SPF provides the capability to use the ***#include*** preprocessing directive, it is possible to use SPF in a collaborative fashion as depicted in Figure 3. For example, the master CMD.spf file could be placed in a centralized location, such as a network shared drive. And, every instance of SPF on the network could set the path of the configuration option SPF_CMD_FILE_PATH to the same value/path, pointing to the master CMD.spf file. Thus, in CMD.spf, you use the ***#include*** directive pointing to your path where you create additional #include SPF files local to your instance and global to every other instance connected to the master CMD.spf file.

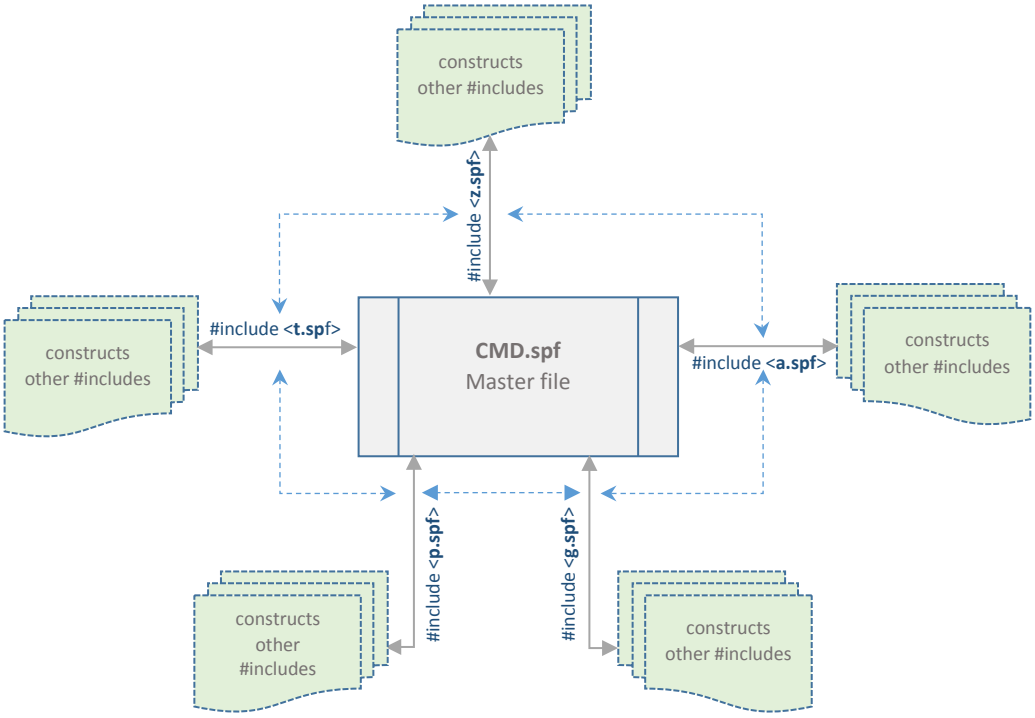

Figure 3 Collaboration Scenario

From there, building connections between different instances of SPF on the network is a matter of configuring the ***#include*** directive to the required path. By doing so, every other instance of SPF has access to every SPF construct defined by any other instance.

EXAMPLES AND SCENARIOS

Using SPF is best illustrated with working examples.

- Example 1
  - Writing a WIN() construct
  - The construct shown below, named **arch**, is responsible for retrieving the OS architecture using WMIC (Windows Management Instrumentation Command-line)
  - This is a very simple construct that demonstrates some of WIN() construct's functionality

```
WIN()
    {
    NAME  = arch
    LOGIC = wmic os get osarchitecture
    INFO  = Get OS Architecture
    TAG   = mokbel
    }
```

  - To execute above construct, all you need to do is typing the name **arch** on the shell

- Example 2
  - Writing an SPF() construct

- The construct shown below, named **alldns**, is responsible for printing all DNS query names and their resolved IPs. It uses the **CALL** operator, calling the WIN construct **pln**
  - **pln** construct is responsible for printing a line number
- This is a very simple construct that demonstrates some of SPF() construct's functionality

```
SPF()
    {
    NAME  = alldns
    LOGIC = -Y (dns && dns.flags.response==1) -Tfields -e dns.qry.name -e
            dns.a -E separator=, | [CALL(pln)]
    INFO  = Get all resolved DNS queries
    TAG   = mokbel|dns
    }
```

- To execute above construct from the shell, first you need to issue the command **setpcap** to specify the name of the pcap you want to work with, and then typing the name **alldns** on the shell
- Alternatively, if the **LOGIC** implementation of **alldns** construct is as follows

```
LOGIC = -Y (dns && dns.flags.response==1) -Tfields -e dns.qry.name -e
            dns.a -E separator=,
```

  - You can mimic the same behavior of the previous **alldns** logic implementation by typing the following set of SPF commands on the shell
    - **alldns op pln**
    - **op** is for opening a pipe (a redirection operator)

ACKNOWLEDGMENT

I would like to thank my colleague Alex Reshetniak for suggesting the clipboard idea with the HELPER CMD ulogic.

Application icon is credited to: Icons made by Freepik from http://www.flaticon.com is licensed by CC 3.0 BY