NAME

etterfilter NG-0.7.3 – Filter compiler for ettercap content filtering engine

SYNOPSIS

etterfilter [*OPTIONS*] *FILE*

DESCRIPTION

The etterfilter utility is used to compile source filter files into binary filter files that can be interpreted by the JIT interpreter in the ettercap(8) filter engine. You have to compile your filter scripts in order to use them in ettercap. All syntax/parse errors will be checked at compile time, so you will be sure to produce a correct binary filter for ettercap.

GENERAL OPTIONS

–o, **––output <FILE>**
you can specify the output file for a source filter file. By default the output is filter.ef.

–t, **––test <FILE>**
you can analyze a compiled filter file with this option. etterfilter will print in a human readable form all the instructions contained in it. It is a sort of "disassembler" for binary filter files.

–d, **––debug**
prints some debug messages during the compilation. Use it more than once to increase the debug level (etterfilter -ddd ...).

–w, **––suppress-warnings**
Don't exit on warnings. With this option the compiler will compile the script even if it contains warnings.

STANDARD OPTIONS

–v, **––version**
Print the version and exit.

–h, **––help**
prints the help screen with a short summary of the available options.

SCRIPTS SYNTAX

A script is a compound of instructions. It is executed sequentially and you can make branches with the 'if' statements. 'if' and 'if/else' statements are the only supported. No loops are implemented. The syntax is almost like C code except that you have to put 'if' blocks into graph parentheses '{' '}', even if they contain only one instruction.

NOTE: you have to put a space between the 'if' and the '('. You must not put the space between the function name and the '('.

Example:
if (conditions) { }
func(args...);

The conditions for an 'if' statement can be either functions or comparisons. Two or more conditions can be linked together with logical operators like OR '||' and AND '&&'.

Example:
```
if (tcp.src == 21 && search(DATA.data, "ettercap")) {
}
```

Pay attention to the operator precedence. You cannot use parentheses to group conditions, so be careful with the order. An AND at the beginning of a conditions block will exclude all the other tests if it is evaluated as false. The parsing is left-to-right, when an operator is found: if it is an AND and the previous condition is false, all the statement is evaluated as false; if it is an OR the parsing goes on even if the condition is false.

Example:
```
if (ip.proto == UDP || ip.proto == TCP && tcp.src == 80) {
}

if (ip.proto == TCP && tcp.src == 80 || ip.proto == UDP) {
}
```

the former condition will match all udp or http traffic. The latter is wrong, because if the packet is not tcp, the whole condition block will be evaluated as false. If you want to make complex conditions, the best way is to split them into nested 'if' blocks.

Every instruction in a block must end with a semicolon ';'.

Comparisons are implemented with the '==' operator and can be used to compare numbers, strings or ip addresses. An ip address MUST be enclosed within two single quotes (eg. '192.168.0.7'). You can also use the 'less than' ('<'), 'greater than' ('>'), 'less or equal' ('<=') and 'greater or equal' ('>=') operators. The lvalue of a comparison must be an offset (see later)

Example:
```
if (DATA.data + 20 == "ettercap" && ip.ttl > 16) {
}
```

Assignments are implemented with the '=' operator and the lvalue can be an offset (see later). The rvalue can be a string, an integer or a hexadecimal value.

Example:
```
ip.ttl = 0xff;
DATA.data + 7 = "ettercap NG";
```

You can also use the 'inc' and 'dec' operations on the packet fields. The operators used are '+=' and '-='. The rvalue can be an integer or a hexadecimal value.

Example:
```
ip.ttl += 5;
```

OFFSET DEFINITION

An offset is identified by a virtual pointer. In short words, an offset is a pointer to the packet buffer. The virtual pointer is a tuple <L, O, S>, where L is the iso/osi level, O is the offset in that level and S is the size of the virtual pointer. You can make algebraic operations on a virtual pointer and the result is still an offset. Specifying 'vp + n' will result in a new virtual pointer <L, O+n, S>. And this is perfectly legal, we have changed the internal offset of that level.

Virtual pointers are in the form 'name.field.subfield'. For example 'ip.ttl' is the virtual pointer for the Time To Live field in the IP header of a packet. It will be translated as <L=3, O=9, S=1>. Indeed it is the 9th byte of level 3 and its size is 1 byte. 'ip.ttl + 1' is the same as 'ip.proto' since the 10th byte of the IP header is the protocol encapsulated in the IP packet.

The list of all supported virtual pointers is in the file etterfilter.tbl. You can add your own virtual pointers by adding a new table or modifying the existing ones. Refer to the comments at the beginning of the file for the syntax of etterfilter.tbl file.

SCRIPTS FUNCTIONS

search(*where*, *what*)

this function searches the string 'what' in the buffer 'where'. The buffer can be either DATA.data or DECODED.data. The former is the payload at layer DATA (ontop TCP or UDP) as it is transmitted on the wire, the latter is the payload decoded/decrypted by dissectors.
So, if you want to search in an SSH connection, it is better to use 'DECODED.data' since 'data' will be encrypted.
The string 'what' can be binary. You have to escape it.

example:
search(DATA.data, "\x41\x42\x43")

regex(*where*, *regex*)

this function will return true if the 'regex' has matched the buffer 'where'. The considerations about 'DECODED.data' and 'DATA.data' mentioned for the function 'search' are the same for the regex function.

NOTE: regex can be used only against a string buffer.

example:
regex(DECODED.data, ".*login.*")

pcre_regex(*where*, *pcre_regex ...)*

this function will evaluate a perl compatible regular expression. You can match against both DATA and DECODED, but if your expression modifies the buffer, it makes sense to operate only on DATA. The function accepts 2 or 3 parameters depending on the operation you want. The two parameter form is used only to match a pattern. The three parameter form means that you want to make a substitution. In both cases, the second parameter is the search string.
You can use $n in the replacement string. These placeholders are referred to the groups created in the search string. (e.g. pcre_regex(DATA.data, "^var1=([:digit:]*)&var2=([:digit:]*)", "var1=$2&var2=$1") will swap the value of var1 and var2).
NOTE: The pcre support is optional in ettercap and will be enabled only if you have the libpcre installed. The compiler will warn you if you try to compile a filter that contains pcre expressions but you don't have libpcre. Use the -w option to suppress the warning.

example:
pcre_regex(DATA.data, ".*foo$")
pcre_regex(DATA.data, "([^]*) bar ([^]*)", "foo $1 $2")

replace(*what*, *with*)

this function replaces the string 'what' with the string 'with'. They can be binary string and must be escaped. The replacement is always performed in DATA.data since is the only payload which gets forwarded. The 'DECODED.data' buffer is used only internally and never reaches the wire.

example:
replace("ethercap", "ettercap")

inject(*what***)**
this function injects the content of the file 'what' after the packet being processed. It always injects in DATA.data. You can use it to replace the entire packet with a fake one using the drop() function right before the inject() command. In that case the filtering engine will drop the current packet and inject the fake one.

example:
inject("./fake_packet")

log(*what***, *where*)**
this function dumps in the file 'where' the buffer 'what'. No information is stored about the packet, only the payload is dumped. So you will see the stream in the file. If you want to log packets in a more enhanced mode, you need to use the ettercap -L option and analyze it with etterlog(8).
The file 'where' must be writable to the user EC_UID (see etter.conf(5)).

example:
log(DECODED.data, "/tmp/interesting.log")

msg(*message***)**
this function displays a message to the user in the User Messages window. It is useful to let the user know whether a particular filter has been successful or not.

example:
msg("Packet filtered successfully")

drop() this function marks the packet "to be dropped". The packet will not be forwarded to the real destination.

example:
drop()

kill() this function kills the connection that owns the matched packet. If it is a TCP connection, a RST is sent to both sides of the connection. If it is an UDP connection, an ICMP PORT UNREACHABLE is sent to the source of the packet.

example:
kill()

exec(*command***)**
this function executes a shell command. You have to provide the full path to the command since it is executed without any environment. There is no way to determine if the command was successful or not. Furthermore, it is executed asynchronously since it is forked by the main process.

example:
exec("/bin/cat /tmp/foo >> /tmp/bar")

exit() this function causes the filter engine to stop executing the code. It is useful to stop the execution of the script on some circumstance checked by an 'if' statement.

example:
exit()

EXAMPLES

Here are some examples of using etterfilter.

etterfilter filter.ecf -o filter.ef

Compiles the source filter.ecf into a binary filter.ef

AUTHORS

Alberto Ornaghi (ALoR) <alor@users.sf.net>
Marco Valleri (NaGA) <naga@antifork.org>

SEE ALSO

etter.filter.examples
ettercap(8) etterlog(8) etter.conf(5) ettercap_curses(8) ettercap_plugins(8)