Creating signatures for ClamAV

1 Introduction

CVD (ClamAV Virus Database) is a digitally signed container that includes signature databases in various text formats. The header of the container is a 512 bytes long string with colon separated fields:

```
ClamAV-VDB:build time:version:number of signatures:functionality
level required:MD5 checksum:digital signature:builder name:build
time (sec)
```

`sigtool --info` displays detailed information about a given CVD file:

```
zolw@localhost:/usr/local/share/clamav$ sigtool -i main.cvd
File: main.cvd
Build time: 09 Dec 2007 15:50 +0000
Version: 45
Signatures: 169676
Functionality level: 21
Builder: sven
MD5: b35429d8d5d60368eea9630062f7c75a
Digital signature: dxsusO/HWP3/GAA7VuZpxYwVsE9b+tCk+tPN6OyjVF/U8
JVh4vYmW8mZ62ZHYMlM903TMZFg5hZIxcjQB3SX0TapdF1SFNzoWjsyH53eXvMDY
eaPVNe2ccXLfEegoda4xU2TezbGfbSEGoU1qolyQYLX674sNA2Ni6l6/CEKYYh
Verification OK.
```

The ClamAV project distributes a number of CVD files, including *main.cvd* and *daily.cvd*.

2 Debug information from libclamav

In order to create efficient signatures for ClamAV it's important to understand how the engine handles input files. The best way to see how it works is having a look

at the debug information from libclamav. You can do it by calling `clamscan` with the `--debug` and `--leave-temps` flags. The first switch makes clamscan display all the interesting information from libclamav and the second one avoids deleting temporary files so they can be analyzed further. The now important part of the info is:

```
$ clamscan --debug attachment.exe
[...]
LibClamAV debug: Recognized MS-EXE/DLL file
LibClamAV debug: Matched signature for file type PE
LibClamAV debug: File type: Executable
```

The engine recognized a windows executable.

```
LibClamAV debug: Machine type: 80386
LibClamAV debug: NumberOfSections: 3
LibClamAV debug: TimeDateStamp: Fri Jan 10 04:57:55 2003
LibClamAV debug: SizeOfOptionalHeader: e0
LibClamAV debug: File format: PE
LibClamAV debug: MajorLinkerVersion: 6
LibClamAV debug: MinorLinkerVersion: 0
LibClamAV debug: SizeOfCode: 0x9000
LibClamAV debug: SizeOfInitializedData: 0x1000
LibClamAV debug: SizeOfUninitializedData: 0x1e000
LibClamAV debug: AddressOfEntryPoint: 0x27070
LibClamAV debug: BaseOfCode: 0x1f000
LibClamAV debug: SectionAlignment: 0x1000
LibClamAV debug: FileAlignment: 0x200
LibClamAV debug: MajorSubsystemVersion: 4
LibClamAV debug: MinorSubsystemVersion: 0
LibClamAV debug: SizeOfImage: 0x29000
LibClamAV debug: SizeOfHeaders: 0x400
LibClamAV debug: NumberOfRvaAndSizes: 16
LibClamAV debug: Subsystem: Win32 GUI
LibClamAV debug: -----------------------------------
LibClamAV debug: Section 0
LibClamAV debug: Section name: UPX0
LibClamAV debug: Section data (from headers - in memory)
LibClamAV debug: VirtualSize: 0x1e000 0x1e000
LibClamAV debug: VirtualAddress: 0x1000 0x1000
LibClamAV debug: SizeOfRawData: 0x0 0x0
```

```
LibClamAV debug: PointerToRawData: 0x400 0x400
LibClamAV debug: Section's memory is executable
LibClamAV debug: Section's memory is writeable
LibClamAV debug: ------------------------------------
LibClamAV debug: Section 1
LibClamAV debug: Section name: UPX1
LibClamAV debug: Section data (from headers - in memory)
LibClamAV debug: VirtualSize: 0x9000 0x9000
LibClamAV debug: VirtualAddress: 0x1f000 0x1f000
LibClamAV debug: SizeOfRawData: 0x8200 0x8200
LibClamAV debug: PointerToRawData: 0x400 0x400
LibClamAV debug: Section's memory is executable
LibClamAV debug: Section's memory is writeable
LibClamAV debug: ------------------------------------
LibClamAV debug: Section 2
LibClamAV debug: Section name: UPX2
LibClamAV debug: Section data (from headers - in memory)
LibClamAV debug: VirtualSize: 0x1000 0x1000
LibClamAV debug: VirtualAddress: 0x28000 0x28000
LibClamAV debug: SizeOfRawData: 0x200 0x1ff
LibClamAV debug: PointerToRawData: 0x8600 0x8600
LibClamAV debug: Section's memory is writeable
LibClamAV debug: ------------------------------------
LibClamAV debug: EntryPoint offset: 0x8470 (33904)
```

The section structure displayed above suggests the executable is packed with UPX.

```
LibClamAV debug: ------------------------------------
LibClamAV debug: EntryPoint offset: 0x8470 (33904)
LibClamAV debug: UPX/FSG/MEW: empty section found - assuming
                 compression
LibClamAV debug: UPX: bad magic - scanning for imports
LibClamAV debug: UPX: PE structure rebuilt from compressed file
LibClamAV debug: UPX: Successfully decompressed with NRV2B
LibClamAV debug: UPX/FSG: Decompressed data saved in
                 /tmp/clamav-90d2d25c9dca42bae6fa9a764a4bcede
LibClamAV debug: ***** Scanning decompressed file *****
LibClamAV debug: Recognized MS-EXE/DLL file
LibClamAV debug: Matched signature for file type PE
```

Indeed, libclamav recognizes the UPX data and saves the decompressed (and rebuilt) executable into `/tmp/clamav-90d2d25c9dca42bae6fa9a764a4bcede`. Then it continues by scanning this new file:

```
LibClamAV debug: File type: Executable
LibClamAV debug: Machine type: 80386
LibClamAV debug: NumberOfSections: 3
LibClamAV debug: TimeDateStamp: Thu Jan 27 11:43:15 2011
LibClamAV debug: SizeOfOptionalHeader: e0
LibClamAV debug: File format: PE
LibClamAV debug: MajorLinkerVersion: 6
LibClamAV debug: MinorLinkerVersion: 0
LibClamAV debug: SizeOfCode: 0xc000
LibClamAV debug: SizeOfInitializedData: 0x19000
LibClamAV debug: SizeOfUninitializedData: 0x0
LibClamAV debug: AddressOfEntryPoint: 0x7b9f
LibClamAV debug: BaseOfCode: 0x1000
LibClamAV debug: SectionAlignment: 0x1000
LibClamAV debug: FileAlignment: 0x1000
LibClamAV debug: MajorSubsystemVersion: 4
LibClamAV debug: MinorSubsystemVersion: 0
LibClamAV debug: SizeOfImage: 0x26000
LibClamAV debug: SizeOfHeaders: 0x1000
LibClamAV debug: NumberOfRvaAndSizes: 16
LibClamAV debug: Subsystem: Win32 GUI
LibClamAV debug: -----------------------------------
LibClamAV debug: Section 0
LibClamAV debug: Section name: .text
LibClamAV debug: Section data (from headers - in memory)
LibClamAV debug: VirtualSize: 0xc000 0xc000
LibClamAV debug: VirtualAddress: 0x1000 0x1000
LibClamAV debug: SizeOfRawData: 0xc000 0xc000
LibClamAV debug: PointerToRawData: 0x1000 0x1000
LibClamAV debug: Section contains executable code
LibClamAV debug: Section's memory is executable
LibClamAV debug: -----------------------------------
LibClamAV debug: Section 1
LibClamAV debug: Section name: .rdata
LibClamAV debug: Section data (from headers - in memory)
LibClamAV debug: VirtualSize: 0x2000 0x2000
LibClamAV debug: VirtualAddress: 0xd000 0xd000
```

```
LibClamAV debug: SizeOfRawData: 0x2000 0x2000
LibClamAV debug: PointerToRawData: 0xd000 0xd000
LibClamAV debug: ------------------------------------
LibClamAV debug: Section 2
LibClamAV debug: Section name: .data
LibClamAV debug: Section data (from headers - in memory)
LibClamAV debug: VirtualSize: 0x17000 0x17000
LibClamAV debug: VirtualAddress: 0xf000 0xf000
LibClamAV debug: SizeOfRawData: 0x17000 0x17000
LibClamAV debug: PointerToRawData: 0xf000 0xf000
LibClamAV debug: Section's memory is writeable
LibClamAV debug: ------------------------------------
LibClamAV debug: EntryPoint offset: 0x7b9f (31647)
LibClamAV debug: Bytecode executing hook id 257 (0 hooks)
attachment.exe: OK
[...]
```

No additional files get created by libclamav. By writing a signature for the decompressed file you have more chances that the engine will detect the target data when it gets compressed with another packer.

This method should be applied to all files for which you want to create signatures. By analyzing the debug information you can quickly see how the engine recognizes and preprocesses the data and what additional files get created. Signatures created for bottom-level temporary files are usually more generic and should help detecting the same malware in different forms.

3 Signature formats

3.1 Hash-based signatures

The easiest way to create signatures for ClamAV is to use filehash checksums, however this method can be only used against static malware.

3.1.1 MD5 hash-based signatures

To create a MD5 signature for `test.exe` use the `--md5` option of sigtool:

```
zolw@localhost:/tmp/test$ sigtool --md5 test.exe > test.hdb
zolw@localhost:/tmp/test$ cat test.hdb
48c4533230e1ae1c118c741c0db19dfb:17387:test.exe
```

That's it! The signature is ready for use:

```
zolw@localhost:/tmp/test$ clamscan -d test.hdb test.exe
test.exe: test.exe FOUND

----------- SCAN SUMMARY -----------
Known viruses: 1
Scanned directories: 0
Engine version: 0.92.1
Scanned files: 1
Infected files: 1
Data scanned: 0.02 MB
Time: 0.024 sec (0 m 0 s)
```

You can change the name (by default sigtool uses the name of the file) and place it inside a `*.hdb` file. A single database file can include any number of signatures. To get them automatically loaded each time clamscan/clamd starts just copy the database file(s) into the local virus database directory (eg. /usr/local/share/clamav).

The hash-based signatures shall not be used for text files, HTML and any other data that gets internally preprocessed before pattern matching. If you really want to use a hash signature in such a case, run clamscan with –debug and –leave-temps flags as described above and create a signature for a preprocessed file left in /tmp. Please keep in mind that a hash signature will stop matching as soon as a single byte changes in the target file.

3.1.2 SHA1 and SHA256 hash-based signatures

ClamAV 0.98 has also added support for SHA1 and SHA256 file checksums. The format is the same as for MD5 file checksum. It can differentiate between them based on the length of the hash string in the signature. For best backwards compatibility, these should be placed inside a `*.hsb` file. The format is:

```
HashString:FileSize:MalwareName
```

3.1.3 PE section based hash signatures

You can create a hash signature for a specific section in a PE file. Such signatures shall be stored inside `.mdb` files in the following format:

```
PESectionSize:PESectionHash:MalwareName
```

The easiest way to generate MD5 based section signatures is to extract target PE sections into separate files and then run sigtool with the option `--mdb`

ClamAV 0.98 has also added support for SHA1 and SHA256 section based signatures. The format is the same as for MD5 PE section based signatures. It can differentiate between them based on the length of the hash string in the signature. For best backwards compatibility, these should be placed inside a `*.msb` file.

3.1.4 Hash signatures with unknown size

ClamAV 0.98 has also added support for hash signatures where the size is not known but the hash is. It is much more performance-efficient to use signatures with specific sizes, so be cautious when using this feature. For these cases, the '*' character can be used in the size field. To ensure proper backwards compatibility with older versions of ClamAV, these signatures must have a minimum functional level of 73 or higher. Signatures that use the wildcard size without this level set will be rejected as malformed.

```
Sample .hsb signature matching any size
HashString:*:MalwareName:73

Sample .msb signature matching any size
*:PESectionHash:MalwareName:73
```

3.2 Body-based signatures

ClamAV stores all body-based signatures in a hexadecimal format. In this section by a hex-signature we mean a fragment of malware's body converted into a hexadecimal string which can be additionally extended using various wildcards.

3.2.1 Hexadecimal format

You can use `sigtool --hex-dump` to convert any data into a hex-string:

```
zolw@localhost:/tmp/test$ sigtool --hex-dump
How do I look in hex?
486f7720646f2049206c6f6f6b20696e206865783f0a
```

3.2.2 Wildcards

ClamAV supports the following wildcards for hex-signatures:

- `??`
  Match any byte.

- `a?`
  Match a high nibble (the four high bits).
  IMPORTANT NOTE: The nibble matching is only available in libclamav with the functionality level 17 and higher therefore please only use it with .ndb signatures followed by ":17" (MinEngineFunctionalityLevel, see **??**).

- `?a`
  Match a low nibble (the four low bits).

- `*`
  Match any number of bytes.

- `{n}`
  Match *n* bytes.

- `{-n}`
  Match *n* or less bytes.

- `{n-}`
  Match *n* or more bytes.

- `{n-m}`
  Match between *n* and *m* bytes ($m > n$).

- `HEXSIG[x-y]aa` or `aa[x-y]HEXSIG`
  Match aa anchored to a hex-signature, see `https://bugzilla.clamav.net/show_bug.cgi?id=776` for discussion and examples.

The range signatures `*` and `{}` virtually separate a hex-signature into two parts, eg. `aabbcc*bbaacc` is treated as two sub-signatures `aabbcc` and `bbaacc` with any number of bytes between them. It's a requirement that each sub-signature includes a block of two static characters somewhere in its body. Note that there is one exception to this restriction; that is when the range wildcard is of the form `{n}` with `n<128`. In this case, ClamAV uses an optimization and translates `{n}` to the string consisting of `n ??` character wildcards. Character wildcards do not divide hex signatures into two parts and so the two static character requirement does not apply.

3.2.3 Character classes

ClamAV supports the following character classes for hex-signatures:

- `(B)`
  Match word boundary (including file boundaries).
- `(L)`
  Match CR, CRLF or file boundaries.
- `(W)`
  Match a non-alphanumeric character.

3.2.4 Alternate strings

- Single-byte alternates (clamav-0.96)
  `(aa|bb|cc|...)` or `!(aa|bb|cc|...)`
  Match a member from a set of bytes [aa, bb, cc, ...].
  - Negation operation can be applied to match any non-member, assumed to be one-byte in length.
  - Signature modifiers and wildcards cannot be applied.
- Multi-byte fixed length alternates
  `(aaaa|bbbb|cccc|...)` or `!(aaaa|bbbb|cccc|...)`
  Match a member from a set of multi-byte alternates [aaaa, bbbb, cccc, ...] of n-length.
  - All set members must be the same length.
  - Negation operation can be applied to match any non-member, assumed to be n-bytes in length (clamav-0.98.2).
  - Signature modifiers and wildcards cannot be applied.
- Generic alternates (clamav-0.99)
  `(alt1|alt2|alt3|...)`
  Match a member from a set of alternates [alt1, alt2, alt3, ...] that can be of variable lengths.
  - Negation operation cannot be applied.
  - Signature modifiers and nibble wildcards [`??, a?, ?a`] can be applied.
  - Ranged wildcards [`{n-m}`] are limited to a fixed range of less than 128 bytes [`{1} -> {127}`].

Note that using signature modifiers and wildcards classifies the alternate type to be a generic alternate. Thus single-byte alternates and multi-byte fixed length alternates can use signature modifiers and wildcards but will be classified as generic alternate. This means that negation cannot be applied in this situation and there is a slight performance impact.

3.2.5 Basic signature format

The simplest (and now deprecated) signature format is:

```
MalwareName=HexSignature
```

ClamAV will scan the entire file looking for HexSignature. All signatures of this type must be placed inside `*.db` files.

3.2.6 Extended signature format

The extended signature format allows for specification of additional information such as a target file type, virus offset or engine version, making the detection more reliable. The format is:

```
MalwareName:TargetType:Offset:HexSignature[:MinFL:[MaxFL]]
```

where `TargetType` is one of the following numbers specifying the type of the target file:

- 0 = any file
- 1 = Portable Executable, both 32- and 64-bit.
- 2 = OLE2 containers, including their specific macros. The OLE2 format is primarily used by MS Office and MSI installation files.
- 3 = HTML (normalized: whitespace transformed to spaces, tags/tag attributes normalized, all lowercase), Javascript is normalized too: all strings are normalized (hex encoding is decoded), numbers are parsed and normalized, local variables/function names are normalized to 'n001' format, argument to eval() is parsed as JS again, unescape() is handled, some simple JS packers are handled, output is whitespace normalized.
- 4 = Mail file
- 5 = Graphics

- 6 = ELF
- 7 = ASCII text file (normalized)
- 8 = Unused
- 9 = Mach-O files
- 10 = PDF files
- 11 = Flash files
- 12 = Java class files

And `Offset` is an asterisk or a decimal number `n` possibly combined with a special modifier:

- `*` = any
- `n` = absolute offset
- `EOF-n` = end of file minus `n` bytes

Signatures for PE, ELF and Mach-O files additionally support:

- `EP+n` = entry point plus n bytes (`EP+0` for `EP`)
- `EP-n` = entry point minus n bytes
- `Sx+n` = start of section `x`'s (counted from 0) data plus `n` bytes
- `SEx` = entire section `x` (offset must lie within section boundaries)
- `SL+n` = start of last section plus `n` bytes

All the above offsets except `*` can be turned into **floating offsets** and represented as `Offset,MaxShift` where `MaxShift` is an unsigned integer. A floating offset will match every offset between `Offset` and `Offset+MaxShift`, eg. `10,5` will match all offsets from 10 to 15 and `EP+n,y` will match all offsets from `EP+n` to `EP+n+y`. Versions of ClamAV older than 0.91 will silently ignore the `MaxShift` extension and only use `Offset`.

Optional `MinFL` and `MaxFL` parameters can restrict the signature to specific engine releases. All signatures in the extended format must be placed inside `*.ndb` files.

3.2.7 Logical signatures

Logical signatures allow combining of multiple signatures in extended format using logical operators. They can provide both more detailed and flexible pattern matching. The logical sigs are stored inside `*.ldb` files in the following format:

```
SignatureName;TargetDescriptionBlock;LogicalExpression;Subsig0;
Subsig1;Subsig2;...
```

where:

- `TargetDescriptionBlock` provides information about the engine and target file with comma separated `Arg:Val` pairs. For args where `Val` is a range, the minimum and maximum values should be expressed as `min-max`.
- `LogicalExpression` specifies the logical expression describing the relationship between `Subsig0...SubsigN`.
  Basis clause: 0,1,...,N decimal indexes are SUB-EXPRESSIONS representing `Subsig0, Subsig1,...,SubsigN` respectively.
  Inductive clause: if `A` and `B` are SUB-EXPRESSIONS and `X, Y` are decimal numbers then `(A&B)`, `(A|B)`, `A=X`, `A=X,Y`, `A>X`, `A>X,Y`, `A<X` and `A<X,Y` are SUB-EXPRESSIONS
- `SubsigN` is n-th subsignature in extended format possibly preceded with an offset. There can be specified up to 64 subsigs.

Keywords used in `TargetDescriptionBlock`:

- `Target:X`: Target file type
- `Engine:X-Y`: Required engine functionality (range; 0.96). Note that if the `Engine` keyword is used, it must be the first one in the `TargetDescriptionBlock` for backwards compatibility
- `FileSize:X-Y`: Required file size (range in bytes; 0.96)
- `EntryPoint`: Entry point offset (range in bytes; 0.96)
- `NumberOfSections`: Required number of sections in executable (range; 0.96)
- `Container:CL_TYPE_*`: File type of the container which stores the scanned file. Specifying `CL_TYPE_ANY` matches on root objects only.

- `Intermediates:CL_TYPE_*>CL_TYPE_*`: File types of intermediate containers which stores the scanned file. Specify 1-16 file types separated by '>' in top-down order ('>' separator not needed for single file type), last type should be the immediate container for the malicious content. `CL_TYPE_ANY` can be used as a wildcard file type. (expr; 0.100.0)

- `IconGroup1`: Icon group name 1 from .idb signature Required engine functionality (range; 0.96)

- `IconGroup2`: Icon group name 2 from .idb signature Required engine functionality (range; 0.96)

Modifiers for subexpressions:

- `A=X`: If the SUB-EXPRESSION A refers to a single signature then this signature must get matched exactly X times; if it refers to a (logical) block of signatures then this block must generate exactly X matches (with any of its sigs).

- `A=0` specifies negation (signature or block of signatures cannot be matched)

- `A=X,Y`: If the SUB-EXPRESSION A refers to a single signature then this signature must be matched exactly X times; if it refers to a (logical) block of signatures then this block must generate X matches and at least Y different signatures must get matched.

- `A>X`: If the SUB-EXPRESSION A refers to a single signature then this signature must get matched more than X times; if it refers to a (logical) block of signatures then this block must generate more than X matches (with any of its sigs).

- `A>X,Y`: If the SUB-EXPRESSION A refers to a single signature then this signature must get matched more than X times; if it refers to a (logical) block of signatures then this block must generate more than X matches and at least Y different signatures must be matched.

- `A<X` and `A<X,Y` as above with the change of "more" to "less".

Examples:

```
Sig1;Target:0;(0&1&2&3)&(4|1);6b6f74656b;616c61;7a6f6c77;7374656
6616e;deadbeef

Sig2;Target:0;((0|1|2)>5,2)&(3|1);6b6f74656b;616c61;7a6f6c77;737
```

```
46566616e

Sig3;Target:0;((0|1|2|3)=2)&(4|1);6b6f74656b;616c61;7a6f6c77;737
46566616e;deadbeef

Sig4;Engine:51-255,Target:1;((0|1)&(2|3))&4;EP+123:33c06834f04100
f2aef7d14951684cf04100e8110a00;S2+78:22??232c2d252229{-15}6e6573
(63|64)61706528;S3+50:68efa311c3b9963cb1ee8e586d32aeb9043e;f9c58
dcf43987e4f519d629b103375;SL+550:6300680065005c0046006900
```

3.2.8 Subsignature Modifiers

ClamAV (clamav-0.99) supports a number of additional subsignature modifiers for logical signatures. This is done by specifying '::' followed by a number of characters representing the desired options. Signatures using subsignature modifiers require `Engine:81-255` for backwards-compatibility.

- Case-Insensitive [`i`]
  Specifying the `i` modifier causes ClamAV to match all alphabetic hex bytes as case-insensitive. All patterns in ClamAV are case-sensitive by default.

- Wide [`w`]
  Specifying the `w` causes ClamAV to match all hex bytes encoded with two bytes per character. Note this simply interweaves each character with NULL characters and does not truly support UTF-16 characters. Wildcards for 'wide' subsignatures are not treated as wide (i.e. there can be an odd number of intermittent characters). This can be combined with `a` to search for patterns in both wide and ascii.

- Fullword [`f`]
  Match subsignature as a fullword (delimited by non-alphanumeric characters).

- Ascii [`a`]
  Match subsignature as ascii characters. This can be combined with `w` to search for patterns in both ascii and wide.

Examples:

```
clamav-nocase-A;Engine:81-255,Target:0;0&1;41414141::i;424242424242::i
    -matches 'AAAA'(nocase) and 'BBBBBB'(nocase)
```

```
clamav-fullword-A;Engine:81-255,Target:0;0&1;414141;68656c6c6f::f
    -matches 'AAA' and 'hello'(fullword)
clamav-fullword-B;Engine:81-255,Target:0;0&1;414141;68656c6c6f::fi
    -matches 'AAA' and 'hello'(fullword nocase)

clamav-wide-B2;Engine:81-255,Target:0;0&1;414141;68656c6c6f::wa
    -matches 'AAA' and 'hello'(wide ascii)
clamav-wide-C0;Engine:81-255,Target:0;0&1;414141;68656c6c6f::iwfa
    -matches 'AAA' and 'hello'(nocase wide fullword ascii)
```

3.3 Special Subsignature Types

3.3.1 Macro subsignatures (clamav-0.96) : ${min-max}MACROID$

Macro subsignatures are used to combine a number of existing extended signatures (.ndb) into a on-the-fly generated alternate string logical signature (.ldb). Signatures using macro subsignatures require Engine:51-255 for backwards-compatibility.

Example:

```
test.ldb:
  TestMacro;Engine:51-255,Target:0;0&1;616161;${6-7}12$

test.ndb:
  D1:0:$12:626262
  D2:0:$12:636363
  D3:0:$30:626264
```

The example logical signature TestMacro is functionally equivalent to:
`TestMacro;Engine:51-255,Target:0;0;616161{3-4}(626262|636363)`

- MACROID points to a group of signatures; there can be at most 32 macro groups.
  - In the example, MACROID is 12 and both D1 and D2 are members of macro group 12. D3 is a member of separate macro group 30.
- {min-max} specifies the offset range at which one of the group signatures should match; the offset range is relative to the starting offset of the preceding subsignature. This means a macro subsignature cannot be the first subsignature.

  - In the example, {min-max} is {6-7} and it is relative to the start of a 616161 match.

- For more information and examples please see https://wwws.clamav.net/bugzilla/show_bug.cgi?id=164.

3.3.2 PCRE subsignatures (clamav-0.99) : Trigger/PCRE/[Flags]

PCRE subsignatures are used within a logical signature (.ldb) to specify regex matches that execute once triggered by a conditional based on preceding subsignatures. Signatures using PCRE subsignatures require Engine:81-255 for backwards-compatibility.

- Trigger is a required field that is a valid LogicalExpression and may refer to any subsignatures that precede this subsignature. Triggers cannot be self-referential and cannot refer to subsequent subsignatures.

- PCRE is the expression representing the regex to execute. PCRE must be delimited by '/' and usage of '/' within the expression need to be escaped. For backward compatibility, ';' within the expression must be expressed as '\x3B'. PCRE cannot be empty and (?UTF*) control sequence is not allowed. If debug is specified, named capture groups are displayed in a post-execution report.

- Flags are a series of characters which affect the compilation and execution of PCRE within the PCRE compiler and the ClamAV engine. This field is optional.

  - g [CLAMAV_GLOBAL] specifies to search for ALL matches of PCRE (default is to search for first match). NOTE: INCREASES the time needed to run the PCRE.

  - r [CLAMAV_ROLLING] specifies to use the given offset as the starting location to search for a match as opposed to the only location; applies to subsigs without maxshifts. By default, in order to facilatate normal ClamAV offset behavior, PCREs are auto-anchored (only attempt match on first offset); using the rolling option disables the auto-anchoring.

  - e [CLAMAV_ENCOMPASS] specifies to CONFINE matching between the specified offset and maxshift; applies only when maxshift is specified. Note: DECREASES time needed to run the PCRE.

  - i [PCRE_CASELESS]

- s [PCRE_DOTALL]
- m [PCRE_MULTILINE]
- x [PCRE_EXTENDED]
- A [PCRE_ANCHORED]
- E [PCRE_DOLLAR_ENODNLY]
- U [PCRE_UNGREEDY]

Examples:

```
Find.All.ClamAV;Engine:81-255,Target:0;1;6265676c6164697427736e6
f7462797465636f6465;0/clamav/g

Find.ClamAV.OnlyAt.299;Engine:81-255,Target:0;2;7374756c747a6765
7473;7063726572656765786c6f6c;299:0&1/clamav/

Find.ClamAV.StartAt.300;Engine:81-255,Target:0;3;616c61696e;6275
6731393238;636c6f736564;300:0&1&2/clamav/r

Find.All.Encompassed.ClamAV;Engine:81-255,Target:0;3;77687961726
56e2774;796f757573696e67;79617261;200,300:0&1&2/clamav/ge

Named.CapGroup.Pcre;Engine:81-255,Target:0;3;636f75727479617264;
616c62756d;74657272696572;50:0&1&2/variable=(?<nilshell>.{16})en
d/gr

Firefox.TreeRange.UseAfterFree;Engine:81-255,Target:0,Engine:81-
255;0&1&2;2e766965772e73656c656374696f6e;2e696e76616c69646174655
3656c656374696f6e;0&1/\x2Eview\x2Eselection.*?\x2Etree\s*\x3D\s*
null.*?\x2Einvalidate/smi

Firefox.IDB.UseAfterFree;Engine:81-255,Target:0;0&1;4944424b6579
52616e6765;0/^\x2e(only|lowerBound|upperBound|bound)\x28.*?\x29.
*?\x2e(lower|upper|lowerOpen|upperOpen)/smi

Firefox.boundElements;Engine:81-255,Target:0;0&1&2;6576656e742e6
26f756e64456c656d656e7473;77696e646f772e636c6f7365;0&1/on(load|c
lick)\s*=\s*\x22?window\.close\s*\x28/si
```

3.4 Icon signatures for PE files

ClamAV 0.96 includes an approximate/fuzzy icon matcher to help detecting malicious executables disguising themselves as innocent looking image files, office documents and the like.

Icon matching is only triggered via .ldb signatures using the special attribute tokens `IconGroup1` or `IconGroup2`. These identify two (optional) groups of icons defined in a .idb database file. The format of the .idb file is:

```
ICONNAME:GROUP1:GROUP2:ICON_HASH
```

where:

- `ICON_NAME` is a unique string identifier for a specific icon,
- `GROUP1` is a string identifier for the first group of icons (`IconGroup1`)
- `GROUP2` is a string identifier for the second group of icons (`IconGroup2`),
- `ICON_HASH` is a fuzzy hash of the icon image

The `ICON_HASH` field can be obtained from the debug output of libclamav. For example:

```
LibClamAV debug: ICO SIGNATURE:
ICON_NAME:GROUP1:GROUP2:18e2e0304ce60a0cc3a09053a30000414100057e
000afe0000e 80006e510078b0a08910d11ad04105e0811510f084e01040c080
a1d0b0021000a39002a41
```

3.5 Signatures for Version Information metadata in PE files

Starting with ClamAV 0.96 it is possible to easily match certain information built into PE files (executables and dynamic link libraries). Whenever you lookup the properties of a PE executable file in windows, you are presented with a bunch of details about the file itself.

These info are stored in a special area of the file resources which goes under the name of `VS_VERSION_INFORMATION` (or versioninfo for short). It is divided into 2 parts. The first part (which is rather uninteresting) is really a bunch of numbers and flags indicating the product and file version. It was originally intended for use with installers which, after parsing it, should be able to determine whether a certain executable or library are to be upgraded/overwritten or are already up

to date. Suffice to say, this approach never really worked and is generally never used.

The second block is much more interesting: it is a simple list of key/value strings, intended for user information and completely ignored by the OS. For example, if you look at ping.exe you can see the company being *"Microsoft Corporation"*, the description *"TCP/IP Ping command"*, the internal name *"ping.exe"* and so on... Depending on the OS version, some keys may be given peculiar visibility in the file properties dialog, however they are internally all the same.

To match a versioninfo key/value pair, the special file offset anchor `VI` was introduced. This is similar to the other anchors (like `EP` and `SL`) except that, instead of matching the hex pattern against a single offset, it checks it against each and every key/value pair in the file. The `VI` token doesn't need nor accept a `+/-` offset like e.g. `EP+1`. As for the hex signature itself, it's just the utf16 dump of the key and value. Only the `??` and `(aa|bb)` wildcards are allowed in the signature. Usually, you don't need to bother figuring it out: each key/value pair together with the corresponding VI-based signature is printed by `clamscan` when the `--debug` option is given.

For example `clamscan --debug freecell.exe` produces:

```
[...]
Recognized MS-EXE/DLL file
in cli_peheader
versioninfo_cb: type: 10, name: 1, lang: 410, rva: 9608
cli_peheader: parsing version info @ rva 9608 (1/1)
VersionInfo (d2de): 'CompanyName'='Microsoft Corporation' -
VI:43006f006d00700061006e0079004e0061006d006500000000004d006900
630072006f0073006f006600740020004300 6f00720070006f0072006100740
069006f006e000000
VersionInfo (d32a): 'FileDescription'='Entertainment Pack
FreeCell Game' - VI:460069006c006500440065007300630072006900700
0740069006f006e000000000045006e007400650072007400610069006e006d
0065006e00740020005000610063006b002000460072006500650043006500 6
c006c002000470061006d0065000000
VersionInfo (d396): 'FileVersion'='5.1.2600.0 (xpclient.010817
-1148)' - VI:460069006c006500560065007200730069006f006e00000000
0035002e0031002e0032003600300030002e003000200028007800700063006
c00690065006e0074002e003000310030003800310037002d00310031003400
380029000000
VersionInfo (d3fa): 'InternalName'='freecell' - VI:49006e007400
650072006e0061006c004e0061006d006500000066007200650065006300650
06c006c000000
```

```
VersionInfo (d4ba): 'OriginalFilename'='freecell' - VI:4f007200
6900670069006e0061006c00460069006c0065006e0061006d0065000000660
072006500650063006500 6c006c000000
VersionInfo (d4f6): 'ProductName'='Sistema operativo Microsoft
Windows' - VI:500072006f006400750063007400 4e0061006d00650000000
000530069007300740065006d00610020006f007000650072006100740069006900
76006f0020004d006900630072006f0073006f0066007400ae0020005700690
06e0064006f0077007300ae000000
VersionInfo (d562): 'ProductVersion'='5.1.2600.0' - VI:50007200
6f006400750063007400560065007200730069006f006e00000035002e00310
02e00320036003000300 02e0030000000
[...]
```

Although VI-based signatures are intended for use in logical signatures you can test them using ordinary `.ndb` files. For example:

```
my_test_vi_sig:1:VI:paste_your_hex_sig_here
```

Final note. If you want to decode a VI-based signature into a human readable form you can use:

```
echo hex_string | xxd -r -p | strings -el
```

For example:

```
$ echo 460069006c006500440065007300630072006900700074006900 6f006e
000000000045006e00740065007200740061006900 6e006d0065006e007400200
05000610063006b002000460072006500650043006500 6c006c00200047006100
6d0065000000 | xxd -r -p | strings -el
FileDescription
Entertainment Pack FreeCell Game
```

3.6 Trusted and Revoked Certificates

Clamav 0.98 checks signed PE files for certificates and verifies each certificate in the chain against a database of trusted and revoked certificates. The signature format is

```
Name;Trusted;Subject;Serial;Pubkey;Exponent;CodeSign;TimeSign;CertSign;
NotBefore;Comment[;minFL[;maxFL]]
```

where the corresponding fields are:

- `Name`: name of the entry
- `Trusted`: bit field, specifying whether the cert is trusted. 1 for trusted. 0 for revoked
- `Subject`: sha1 of the Subject field in hex
- `Serial`: the serial number as clamscan –debug –verbose reports
- `Pubkey`: the public key in hex
- `Exponent`: the exponent in hex. Currently ignored and hardcoded to 010001 (in hex)
- `CodeSign`: bit field, specifying whether this cert can sign code. 1 for true, 0 for false
- `TimeSign`: bit field. 1 for true, 0 for false
- `CertSign`: bit field, specifying whether this cert can sign other certs. 1 for true, 0 for false
- `NotBefore`: integer, cert should not be added before this variable. Defaults to 0 if left empty
- `Comment`: comments for this entry

The signatures for certs are stored inside `.crb` files.

3.7 Signatures based on container metadata

ClamAV 0.96 allows creating generic signatures matching files stored inside different container types which meet specific conditions. The signature format is

```
VirusName:ContainerType:ContainerSize:FileNameREGEX:
FileSizeInContainer:FileSizeReal:IsEncrypted:FilePos:
Res1:Res2[:MinFL[:MaxFL]]
```

where the corresponding fields are:

- `VirusName`: Virus name to be displayed when signature matches

- `ContainerType`: one of `CL_TYPE_ZIP`, `CL_TYPE_RAR`, `CL_TYPE_ARJ`, `CL_TYPE_MSCAB`, `CL_TYPE_7Z`, `CL_TYPE_MAIL`, `CL_TYPE_(POSIX|OLD)_TAR`, `CL_TYPE_CPIO_(OLD|ODC|NEWC|CRC)` or `*` to match any of the container types listed here
- `ContainerSize`: size of the container file itself (eg. size of the zip archive) specified in bytes as absolute value or range `x-y`
- `FileNameREGEX`: regular expression describing name of the target file
- `FileSizeInContainer`: usually compressed size; for MAIL, TAR and CPIO == `FileSizeReal`; specified in bytes as absolute value or range
- `FileSizeReal`: usually uncompressed size; for MAIL, TAR and CPIO == `FileSizeInContainer`; absolute value or range
- `IsEncrypted`: 1 if the target file is encrypted, 0 if it's not and `*` to ignore
- `FilePos`: file position in container (counting from 1); absolute value or range
- `Res1`: when `ContainerType` is `CL_TYPE_ZIP` or `CL_TYPE_RAR` this field is treated as a CRC sum of the target file specified in hexadecimal format; for other container types it's ignored
- `Res2`: not used as of ClamAV 0.96

The signatures for container files are stored inside `.cdb` files.

3.8 Signatures based on ZIP/RAR metadata (obsolete)

The (now obsolete) archive metadata signatures can be only applied to ZIP and RAR files and have the following format:

```
virname:encrypted:filename:normal size:csize:crc32:cmethod:
fileno:max depth
```

where the corresponding fields are:

- Virus name
- Encryption flag (1 – encrypted, 0 – not encrypted)
- File name (this is a regular expression - * to ignore)
- Normal (uncompressed) size (* to ignore)

- Compressed size (* to ignore)
- CRC32 (* to ignore)
- Compression method (* to ignore)
- File position in archive (* to ignore)
- Maximum number of nested archives (* to ignore)

The database file should have the extension of `.zmd` or `.rmd` for zip or rar metadata respectively.

3.9 Whitelist databases

To whitelist a specific file use the MD5 signature format and place it inside a database file with the extension of `.fp`. To whitelist a specific file with the SHA1 or SHA256 file hash signature format, place the signature inside a database file with the extension of `.sfp`.

To whitelist a specific signature from the database you just add its name into a local file called local.ign2 stored inside the database directory. You can additionally follow the signature name with the MD5 of the entire database entry for this signature, eg:

```
Eicar-Test-Signature:bc356bae4c42f19a3de16e333ba3569c
```

In such a case, the signature will no longer be whitelisted when its entry in the database gets modified (eg. the signature gets updated to avoid false alerts).

3.10 Signature names

ClamAV uses the following prefixes for signature names:

- *Worm* for Internet worms
- *Trojan* for backdoor programs
- *Adware* for adware
- *Flooder* for flooders
- *HTML* for HTML files
- *Email* for email messages

- *IRC* for IRC trojans
- *JS* for Java Script malware
- *PHP* for PHP malware
- *ASP* for ASP malware
- *VBS* for VBS malware
- *BAT* for BAT malware
- *W97M*, *W2000M* for Word macro viruses
- *X97M*, *X2000M* for Excel macro viruses
- *O97M*, *O2000M* for generic Office macro viruses
- *DoS* for Denial of Service attack software
- *DOS* for old DOS malware
- *Exploit* for popular exploits
- *VirTool* for virus construction kits
- *Dialer* for dialers
- *Joke* for hoaxes

Important rules of the naming convention:

- always use a -zippwd suffix in the malware name for signatures of type zmd,
- always use a -rarpwd suffix in the malware name for signatures of type rmd,
- only use alphanumeric characters, dash (-), dot (.), underscores (_) in malware names, never use space, apostrophe or quote mark.

3.11 Using YARA rules in ClamAV

ClamAV version 0.99 and above can process YARA rules. ClamAV virus database file names ending with ".yar" or ".yara" are parsed as yara rule files. The link to the YARA rule grammar documentation may be found at http://plusvic.github.io/yara/. There are currently a few limitations on using YARA rules within ClamAV:

- YARA modules are not yet supported by ClamAV. This includes the "import" keyword and any YARA module-specific keywords.
- Global rules("global" keyword) are not supported by ClamAV.
- External variables("contains" and "matches" keywords) are not supported.
- YARA rules pre-compiled with the *yarac* command are not supported.
- As in the ClamAV logical and extended signature formats, YARA strings and segments of strings separated by wild cards must represent at least two octets of data.
- There is a maximum of 64 strings per YARA rule.
- YARA rules in ClamAV must contain at least one literal, hexadecimal, or regular expression string.

In addition, there are a few more ClamAV processing modes that may affect the outcome of YARA rules.

- *File decomposition and decompression* - Since ClamAV uses file decomposition and decompression to find viruses within de-archived and uncompressed inner files, YARA rules executed by ClamAV will match against these files as well.
- *Normalization* - By default, ClamAV normalizes HTML, JavaScript, and ASCII text files. YARA rules in ClamAV will match against the normalized result. The effects of normalization of these file types may be captured using `clamscan --leave-temps --tempdir=mytempdir`. YARA rules may then be written using the normalized file(s) found in `mytempdir`. Alternatively, starting with ClamAV 0.100.0, `clamscan --normalize=no` will prevent normalization and only scan the raw file. To obtain similar behavior prior to 0.99.2, use `clamscan --scan-html=no`. The corresponding parameters for clamd.conf are `Normalize` and `ScanHTML`.

- *YARA conditions driven by string matches* - All YARA conditions are driven by string matches in ClamAV. This saves from executing every YARA rule on every file. Any YARA condition may be augmented with a string match clause which is always true, such as:

```
rule CheckFileSize
{
  strings:
    $abc = "abc"
  condition:
    ($abc or not $abc) and filesize < 200KB
}
```

  This will ensure that the YARA condition always performs the desired action (checking the file size in this example),

3.12 Passwords for archive files [experimental]

ClamAV 0.99 allows for users to specify password attempts for certain password-compatible archives. Passwords will be attempted in order of appearance in the password signature file which use the extension of `.pwdb`. If no passwords apply or none are provided, ClamAV will default to the original behavior of parsing the file. Currently, as of ClamAV 0.99 [flevel 81], only `.zip` archives using the traditional PKWARE encryption are supported. The signature format is

`SignatureName;TargetDescriptionBlock;PWStorageType;Password`

where:

- `SignatureName`: name to be displayed during debug when a password is successful
- `TargetDescriptionBlock`: provides information about the engine and target file with comma separated Arg:Val pairs
  - `Engine:X-Y`: Required engine functionality
  - `Container:CL_TYPE_*`: File type of applicable containers
- `PWStorageType`: determines how the password field is parsed
  - 0 = cleartext
  - 1 = hex

- `Password`: value used in password attempt

The signatures for password attempts are stored inside `.pwdb` files.

4 Special files

4.1 HTML

ClamAV contains a special HTML normalisation code which helps to detect HTML exploits. Running `sigtool --html-normalise` on a HTML file should generate the following files:

- nocomment.html - the file is normalized, lower-case, with all comments and superfluous white space removed
- notags.html - as above but with all HTML tags removed

The code automatically decodes JScript.encode parts and char ref's (e.g. `&#102;`). You need to create a signature against one of the created files. To eliminate potential false positive alerts the target type should be set to 3.

4.2 Text files

Similarly to HTML all ASCII text files get normalized (converted to lower-case, all superfluous white space and control characters removed, etc.) before scanning. Use `clamscan --leave-temps` to obtain a normalized file then create a signature with the target type 7.

4.3 Compressed Portable Executable files

If the file is compressed with UPX, FSG, Petite or other PE packer supported by libclamav, run `clamscan` with `--debug --leave-temps`. Example output for a FSG compressed file:

```
LibClamAV debug: UPX/FSG/MEW: empty section found - assuming compression
LibClamAV debug: FSG: found old EP @119e0
LibClamAV debug: FSG: Unpacked and rebuilt executable saved in
/tmp/clamav-f592b20f9329ac1c91f0e12137bcce6c
```

Next create a type 1 signature for `/tmp/clamav-f592b20f9329ac1c91f0e12137bcce6c`