ClamAV Bytecode Compiler

User Manual

Contents

CHAPTER 1
Installation

1.1. Requirements

The ClamAV Bytecode Compiler uses the LLVM compiler framework, thus requires an Operating System where building LLVM is supported:

- FreeBSD/x86
- Linux/{x86,x86_64,ppc}
- Mac OS X/{x86,ppc}
- Solaris/sparcv9
- Windows/x86 using mingw32 or Visual Studio

The following packages are required to compile the ClamAV Bytecode Compiler:

- GCC C and C++ compilers (minimum 4.1.3, recommended: 4.3.4 or newer) [1].
- Perl (version 5.6.0+)
- GNU make (version 3.79+, recommended 3.81)

The following packages are optional, but highly recommended:

- Python (version 2.5.4+?) - for running the tests

1.2. Obtaining the ClamAV Bytecode Compiler

You can obtain the source code in one of the following ways [2]

- Check out the source code using git native protocol:

  ```
  git clone git://git.clamav.net/git/clamav-bytecode-compiler
  ```

- Check out the source code using HTTP:

  ```
  git clone http://git.clamav.net/git/clamav-bytecode-compiler.git
  ```

You can keep the source code updated using:

```
git pull
```

[1] Note that several versions of GCC have bugs when compiling LLVM, see http://llvm.org/docs/GettingStarted.html#brokengcc for a full list. Also LLVM requires support for atomic builtins for multithreaded mode, which gcc 3.4.x doesn't have

[2] For now the use the internal clamtools repository:
git clone username@git.clam.sourcefire.com:/var/lib/git/clamtools.git

1.3. Building

1.3.1. Disk space

A minimalistic release build requires 100M of disk space.

Testing the compiler requires a full build, 320M of disk space. A debug build requires significantly more disk space (1.4G for a minimalistic debug build).

Note that this only needed during the build process, once installed only 12M is needed.

1.3.2. Create build directory

Building requires a separate object directory, building in the source directory is not supported. Create a build directory:

```
$ cd clamav-bytecode-compiler && mkdir obj
```

Run configure (you can use any prefix you want, this example uses `/usr/local/clamav`):

```
$ cd obj && ../llvm/configure --enable-optimized \
 --enable-targets=host-only --disable-bindings \
--prefix=/usr/local/clamav
```

Run the build under ulimit [1]:

```
$ (ulimit -t 3600 -v 512000 && make clambc-only -j4)
```

1.4. Testing

```
$ (ulimit -t 3600 v 512000 && make -j4)
$ make check-all
```

If make check reports errors, check that your compiler is NOT on this list: http://llvm.org/docs/GettingStarted.html#brokengcc.

If it is, then your compiler is buggy, and you need to do one of the following: upgrade your compiler to a non-buggy version, upgrade the OS to one that has a non-buggy compiler, compile with `export OPTMIZE_OPTION=-O2`, or `export OPTIMIZE_OPTION=-O1`, or `export OPTIMIZE_OPTION=\-O1`.

If not you probably found a bug, report it at http://bugs.clamav.net

1.5. Installing

Install it:

```
$ make install-clambc -j8
```

1.5.1. Structure of installed files

1. The ClamAV Bytecode compiler driver: `$PREFIX/bin/clambc-compiler`
2. ClamAV bytecode header files:

   ```
   $PREFIX/lib/clang/1.1/include:
   bcfeatures.h
   bytecode_{api_decl.c,api,disasm,execs,features}.h
   bytecode.h
   bytecode_{local,pe,types}.h
   ```

3. clang compiler (with ClamAV bytecode backend) compiler include files:

[1] compiling some files can be very memory intensive, especially with older compilers

```
$PREFIX/lib/clang/1.1/include:
emmintrin.h
float.h
iso646.h
limits.h
{,p,t,x}mmintrin.h
mm_malloc.h
std{arg,bool,def,int}.h
tgmath.h
```

4. User manual

```
$PREFIX/docs/clamav/clambc-user.pdf
```

CHAPTER 2
Tutorial

2.1. Short introduction to the bytecode language

2.1.1. Types, variables and constants

2.1.2. Arrays and pointers

2.1.3. Arithmetics

2.1.4. Functions

2.1.5. Control flow

2.1.6. Common functions

2.2. Writing logical signature bytecodes

[1] Logical signatures can be used as triggers for executing bytecode. However, instead of describing a logical signature as a `.ldb` pattern, you use (simple) C code which is later translated to a `.ldb`-style logical signature by the ClamAV Bytecode Compiler.

A bytecode triggered by a logical signature is much more powerful than a logical signature itself: you can write complex algorithmic detections, and use the logical signature as a *filter* (to speed up matching). Thus another name for "logical signature bytecodes" is "algorithmic detection bytecodes". The detection you write in bytecode has read-only access to the file being scanned and its metadata (PE sections, EP, etc.).

2.2.1. Structure of a bytecode for algorithmic detection

Algorithmic detection bytecodes are triggered when a logical signature matches. They can execute an algorithm that determines whether the file is infected and with which virus.

A bytecode can be either algorithmic or an unpacker (or other hook), but not both.

It consists of:

- Definition of virusnames used in the bytecode
- Pattern definitions (for logical subexpressions)
- The logical signature as C function: `bool logical_trigger(void)`
- The `int entrypoint(void)` function which gets executed when the logical signature matches
- (Optional) Other functions and global constants used in `entrypoint`

The syntax for defining logical signatures, and an example is described in Section 2.2.4.

The function `entrypoint` must report the detected virus by calling `foundVirus` and returning `0`. It is recommended that you always return `0`, otherwise a warning is shown and the file is considered clean. If `foundVirus` is not called, then ClamAV also assumes the file is clean.

[1] See Section 4.3 for more details about logical signatures in bytecode.

2.2.2. Virusnames

Each logical signature bytecode must have a virusname prefix, and one or more virusnames. The virusname prefix is used by the SI to ensure unique virusnames (a unique number is appended for duplicate prefixes).

Program 1 Declaring virusnames

```
/* Prefix, used for duplicate detection and fixing */
VIRUSNAME_PREFIX("Trojan.Foo")
/* You are only allowed to set these virusnames as found */
VIRUSNAMES("A", "B")
/* File type */
TARGET(2)
```

In Program 1 3 predefied macros are used:

- `VIRUSNAME_PREFIX` which must have exactly one string argument
- `VIRUSNAMES` which must have one or more string arguments
- `TARGET` which must have exactly one integer argument

In this example, the bytecode could generate one of these virusnames: `Trojan.Foo.A`, or `Trojan.Foo.B`, by calling `foundVirus("A")` or `foundVirus("B")` respectively (notice that the prefix is not part of these calls).

2.2.3. Patterns

Logical signatures use `.ndb` style patterns, an example on how to define these is shown in Program 2.

Program 2 Declaring patterns

```
SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(magic)
DECLARE_SIGNATURE(check)
DECLARE_SIGNATURE(zero)
SIGNATURES_DECL_END

SIGNATURES_DEF_BEGIN
DEFINE_SIGNATURE(magic, "EP+0:aabb")
DEFINE_SIGNATURE(check, "f00d")
DEFINE_SIGNATURE(zero, "ffff")
SIGNATURES_END
```

Each pattern has a name (like a variable), and a string that is the hex pattern itself. The declarations are delimited by the macros `SIGNATURES_DECL_BEGIN`, and `SIGNATURES_DECL_END`. The definitions are delimited by the macros `SIGNATURES_DEF_BEGIN`, and `SIGNATURES_END`. Declarations must always come before definitions, and you can have only one declaration and declaration section! (think of declaration like variable declarations, and definitions as variable assignments, since that what they are under the hood). The order in which you declare the signatures is the order in which they appear in the generated logical signature.

You can use any name for the patterns that is a valid record field name in C, and doesn't conflict with anything else declared.

After using the above macros, the global variable `Signatures` will have two new fields: `magic`, and `zero`. These can be used as arguments to the functions `count_match()`, and `matches()` anywhere in the program as shown in Program 3:

- `matches(Signatures.match)` will return true when the `match` signature matches (at least once)
- `count_match(Signatures.zero)` will return the number of times the `zero` signature matched
- `count_match(Signatures.check)` will return the number of times the `check` signature matched

The condition in the `if` can be interpreted as: if the `match` signature has matched at least once, and the number of times the `zero` signature matched is higher than the number of times the `check` signature matched, then we have found a virus `A`, otherwise the file is clean.

Program 3 Using patterns

```
int entrypoint(void)
{
  if (matches(Signatures.match) && count_match(Signatures.zero) >
      count_match(Signatures.check))
    foundVirus("A");
  return 0;
}
```

2.2.4. Single subsignature

The simplest logical signature is like a `.ndb` signature: a virus name, signature target, 0 as logical expression [1], and a `ndb`-style pattern.

The code for this is shown in Program 4

Program 4 Single subsignature example

```
/* Declare the prefix of the virusname */
VIRUSNAME_PREFIX("Trojan.Foo")
/* Declare the suffix of the virusname */
VIRUSNAMES("A")
/* Declare the signature target type (1 = PE) */
TARGET(1)

/* Declare the name of all subsignatures used */
SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(magic)
SIGNATURES_DECL_END

/* Define the pattern for each subsignature */
SIGNATURES_DEF_BEGIN
DEFINE_SIGNATURE(magic, "aabb")
SIGNATURES_END

/* All bytecode triggered by logical signatures must have this
   function */
bool logical_trigger(void)
{
  /* return true if the magic subsignature matched,
   * its pattern is defined above to "aabb" */
  return count_match(Signatures.magic) != 2;
}

/* This is the bytecode function that is actually executed when the logical
 * signature matched */
int entrypoint(void)
{
  /* call this function to set the suffix of the virus found */
  foundVirus("A");
  /* success, return 0 */
  return 0;
}
```

The logical signature (created by the compiler) looks like this: `Trojan.Foo.{A};Target:2;0;aabb`

Of course you should use a `.ldb` signature in this case when all the processing in `entrypoint` is only setting a virusname and returning. However, you can do more complex checks in `entrypoint`, once the bytecode was triggered by the `logical_trigger`

In the example in Program 4 the pattern was used without an anchor; such a pattern matches at any offset. You can use offsets though, the same way as in `.ndb` signatures, see Program 5 for an example.

[1] meaning that subexpression 0 must match

2.2.5. Multiple subsignatures

An example for this is shown in Program 5. Here you see the following new features used: [1]

- Multiple virusnames returned from a single bytecode (with common prefix)
- Multiple subsignatures, each with a name of your choice
- A pattern with an anchor (`EP+0:aabb`)
- More subsignatures defined than used in the logical expression

The logical signature looks like this:

```
Trojan.Foo.{A,B};Target:2;(((0|1|2)=42,2)|(3=10));EP+0:aabb;ffff;aaccee;f00d;dead
```

Notice how the subsignature that is not used in the logical expression (number 4, `dead`) is used in `entrypoint` to decide the virus name. This works because ClamAV does collect the match counts for all subsignatures (regardless if they are used or not in a signature). The `count_match(Signatures.check2)` call is thus a simple memory read of the count already determined by ClamAV.

Also notice that comments can be used freely: they are ignored by the compiler. You can use either C-style multiline comments (start comment with `/*`, end with `*/`), or C++-style single-line comments (start comment with `//`, automatically ended by newline).

2.2.6. W32.Polipos.A detector rewritten as bytecode

2.2.7. Virut detector in bytecode

2.3. Writing regular expressions in bytecode

ClamAV only supports a limited set of regular expressions in `.ndb format` : wildcards. The bytecode compiler allows you to compile fully generic regular expressions to bytecode directly. When libclamav loads the bytecode, it will compile to native code (if using the JIT), so it should offer quite good performance.

The compiler currently uses `re2c` to compile regular expressions to C code, and then compile that to bytecode. The internal workings are all transparent to the user: the compiler automatically uses `re2c` when needed, and `re2c` is embedded in the compiler, so you don't need to install it.

The syntax of regular expressions are similar to the one used by `POSIX regular expressions`, except you have to quote literals, since unquoted they are interpreted as regular expression names.

2.3.1. A very simple regular expression

Lets start with a simple example, to match this POSIX regular expression: `eval([a-zA-Z_][a-zA-Z0-9_]*\.unescape`.

See Program 6 [2].

There are several new features introduced here, here is a step by step breakdown:

`REGEX_SCANNER` this declares the data structures needed by the regular expression matcher

`seek(0, SEEK_SET)` this sets the current file offset to position `0`, matching will start at this position. For offset 0 it is not strictly necessary to do this, but it serves as a reminder that you might want to start matching somewhere, that is not necessarily 0.

`for(;;) { REGEX_LOOP_BEGIN` this creates the regular expression matcher main loop. It takes the current file byte-by-byte [3] and tries to match one of the regular expressions.

`/*!re2c` This mark the beginning of the regular expression description. The entire regular expression block is a C comment, starting with `!re2c`

`ANY = [^];` This declares a regular expression named `ANY` that matches any byte.

`"eval("[a-zA-Z_][a-zA-Z_0-9]*".unescape" {` This is the actual regular expression.

[1] In case of a duplicate virusname the prefix is appended a unique number by the SI

[2] This omits the virusname, and logical signature declarations

[3] it is not really reading byte-by-byte, it is using a buffer to speed things up

Program 5 Multiple subsignatures

```
/* You are only allowed to set these virusnames as found */
VIRUSNAME_PREFIX("Test")
VIRUSNAMES("A", "B")
TARGET(1)

SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(magic)
DECLARE_SIGNATURE(zero)
DECLARE_SIGNATURE(check)
DECLARE_SIGNATURE(fivetoten)
DECLARE_SIGNATURE(check2)
SIGNATURES_DECL_END

SIGNATURES_DEF_BEGIN
DEFINE_SIGNATURE(magic, "EP+0:aabb")
DEFINE_SIGNATURE(zero, "ffff")
DEFINE_SIGNATURE(fivetoten, "aaccee")
DEFINE_SIGNATURE(check, "f00d")
DEFINE_SIGNATURE(check2, "dead")
SIGNATURES_END

bool logical_trigger(void)
{
    unsigned sum_matches = count_match(Signatures.magic)+
        count_match(Signatures.zero) + count_match(Signatures.fivetoten);
    unsigned unique_matches = matches(Signatures.magic)+
            matches(Signatures.zero)+ matches(Signatures.fivetoten);
    if (sum_matches == 42 && unique_matches == 2) {
        // The above 3 signatures have matched a total of 42 times, and at least
        // 2 of them have matched
        return true;
    }
    // If the check signature matches 10 times we still have a match
    if (count_match(Signatures.check) == 10)
        return true;
    // No match
    return false;
}

int entrypoint(void)
{
    unsigned count = count_match(Signatures.check2);
    if (count >= 2)
//      foundVirus(count == 2 ? "A" : "B");
      if (count == 2)
        foundVirus("A");
      else
        foundVirus("B");
    return 0;
}
```

Program 6 Simple regular expression example

```
int entrypoint(void)
{
    REGEX_SCANNER;
    seek(0, SEEK_SET);
    for (;;) {
        REGEX_LOOP_BEGIN

          /* !re2c
             ANY = [^];

             "eval("[a-zA-Z_][a-zA-Z_0-9]*".unescape" {
                long pos = REGEX_POS;
                if (pos < 0)
                  continue;
                debug("unescape found at:");
                debug(pos);
             }
             ANY { continue; }
          */
    }
    return 0;
}
```

`"eval("` This matches the literal string `eval(`. Literals have to be placed in double quotes `"` here, unlike in POSIX regular expressions or PCRE. If you want case-insensitive matching, you can use `'`.

`[a-zA-Z_]` This is a character class, it matches any lowercase, uppercase or `_` characters.

`[a-zA-Z_0-9]*"` Same as before, but with repetition. `*` means match zero or more times, `+` means match one or more times, just like in POSIX regular expressions.

`".unescape"` A literal string again

`{` start of the *action* block for this regular expression. Whenever the regular expression matches, the attached C code is executed.

`long pos = REGEX_POS;` this determines the absolute file offset where the regular expression has matched. Note that because the regular expression matcher uses a buffer, using just `seek(0, SEEK_CUR)` would give the current position of the end of that buffer, and not the current position during regular expression matching. You have to use the `REGEX_POS` macro to get the correct position.

`debug(...)` Shows a debug message about what was found and where. This is extremely helpful when you start writing regular expressions, and nothing works: you can determine whether your regular expression matched at all, and if it matched where you thought it would. There is also a `DEBUG_PRINT_MATCH` that prints the entire matched string to the debug output. Of course before publishing the bytecode you might want to turn off these debug messages.

`}` closes the *action* block for this regular expression

`ANY { continue; }` If none of the regular expressions matched so far, just keep running the matcher, at the next byte

`*/` closes the regular expression description block

`}` closes the `for()` loop

You may have multiple regular expressions, or declare multiple regular expressions with a name, and use those names to build more complex regular expressions.

2.3.2. Named regular expressions

2.4. Writing unpackers

2.4.1. Structure of a bytecode for unpacking (and other hooks)

When writing an unpacker, the bytecode should consist of:

- Define which hook you use (for example `PE_UNPACKER_DECLARE` for a PE hook)
- An `int entrypoint(void)` function that reads the current file and unpacks it to a new file
- Return 0 from `entrypoint` if you want the unpacked file to be scanned
- (Optional) Other functions and global constants used by `entrypoint`

2.4.2. Detecting clam.exe via bytecode

Example provided by aCaB:

2.4.3. Detecting clam.exe via bytecode (disasm)

Example provided by aCaB:

2.4.4. A simple unpacker

2.4.5. Matching PDF javascript

2.4.6. YC unpacker rewritten as bytecode

CHAPTER 3
Usage

3.1. Invoking the compiler

Compiling is similar to gcc [1]:

```
$ /usr/local/clamav/bin/clambc-compiler foo.c -o foo.cbc -O2
```

This will compile the file `foo.c` into a file called `foo.cbc`, that can be loaded by ClamAV, and packed inside a `.cvd` file.

The compiler by default has all warnings turned on.

Supported optimization levels: `-O0, -O1, -O2, -O3`. [2] It is recommended that you always compile with at least `-O1`.

Warning options: `-Werror` (transforms all warnings into errors).

Preprocessor flags:

-I <directory> Searches in the given directory when it encounters a `#include "headerfile"` directive in the source code, in addition to the system defined header search directories.

-D <MACRONAME>=<VALUE> Predefine given `<MACRONAME>` to be equal to `<VALUE>`.

-U <MACRONAME> Undefine a predefined macro

The compiler also supports some other commandline options (see `clambc-compiler --help` for a full list), however some of them have no effect when using the ClamAV bytecode backend (such as the X86 backend options). You shouldn't need to use any flags not documented above.

3.1.1. Compiling C++ files

Filenames with a `.cpp` extension are compiled as C++ files, however `clang++` is not yet ready for production use, so this is EXPERIMENTAL currently. For now write bytecodes in C.

3.2. Running compiled bytecode

After compiling a C source file to bytecode, you can load it in ClamAV:

3.2.1. ClamBC

ClamBC is a tool you can use to test whether the bytecode loads, compiles, and can execute its entrypoint successfully. Usage:

```
clambc <file> [function] [param1 ...]
```

For example loading a simple bytecode with 2 functions is done like this:

[1] Note that the ClamAV bytecode compiler will refuse to compile code it considers insecure

[2] Currently -O0 doesn't work

```
$ clambc foo.cbc
LibClamAV debug: searching for unrar, user-searchpath: /usr/local/lib
LibClamAV debug: unrar support loaded from libclamunrar_iface.so.6.0.4 libclamunrar_iface_so_6_0
LibClamAV debug: bytecode: Parsed 0 APIcalls, maxapi 0
LibClamAV debug: Parsed 1 BBs, 2 instructions
LibClamAV debug: Parsed 1 BBs, 2 instructions
LibClamAV debug: Parsed 2 functions
Bytecode loaded
Running bytecode function :0
Bytecode run finished
Bytecode returned: 0x8
Exiting
```

3.2.2. clamscan, clamd

You can tell clamscan to load the bytecode as a database directly:

```
$ clamscan -dfoo.cbc
```

Or you can instruct it to load all databases from a directory, then clamscan will load all supported formats, including files with bytecode, which have the `.cbc` extension.

```
$ clamscan -ddirectory
```

You can also put the bytecode files into the default database directory of ClamAV (usually `/usr/local/share/clamav`) to have it loaded automatically from there. Of course, the bytecode can be stored inside CVD files, too.

3.3. Debugging bytecode

3.3.1. "printf" style debugging

Printf, and printf-like format specifiers are not supported in the bytecode. You can use these functions instead of printf to print strings and integer to clamscan's –debug output:

debug_print_str, debug_print_uint, debug_print_str_start, debug_print_str_nonl.

You can also use the `debug` convenience wrapper that automatically prints as string or integer depending on parameter type: debug, debug, debug.

See Program 7 for an example.

3.3.2. Single-stepping

If you have GDB 7.0 (or newer) you can single-step [1] [2] during the execution of the bytecode.

- Run clambc or clamscan under gdb:

  ```
  $ ./libtool --mode=execute gdb clamscan/clamscan
  ...
  (gdb) b cli_vm_execute_jit
  Are you sure ....? y
  (gdb) run -dfoo.cbc
  ...
  Breakpoint ....

  (gdb) step
  (gdb) next
  ```

  You can single-step through the execution of the bytecode, however you can't (yet) print values of individual variables, you'll need to add debug statements in the bytecode to print interesting values.

[1] not yet implemented in libclamav
[2] assuming you have JIT support

Program 7 Example of using debug APIs

```
/* test debug APIs */
int entrypoint(void)
{
  /* print a debug message, followed by newline */
  debug_print_str("bytecode started", 16);

  /* start a new debug message, don't end with newline yet */
  debug_print_str_start("Engine functionality level: ", 28);
  /* print an integer, no newline */
  debug_print_uint(engine_functionality_level());
  /* print a string without starting a new debug message, and without
   * terminating with newline */
  debug_print_str_nonl(", dconf functionality level: ", 28);
  debug_print_uint(engine_dconf_level());
  debug_print_str_nonl("\n", 1);
  debug_print_str_start("Engine scan options: ", 21);
  debug_print_uint(engine_scan_options());
  debug_print_str_nonl(", db options: ", 13);
  debug_print_uint(engine_db_options());
  debug_print_str_nonl("\n", 1);

  /* convenience wrapper to just print a string */
  debug("just print a string");
  /* convenience wrapper to just print an integer */
  debug(4);
  return 0xf00d;
}
```

CHAPTER 4
ClamAV bytecode language

The bytecode that ClamAV loads is a simplified form of the LLVM Intermediate Representation, and as such it is language-independent.

However currently the only supported language from which such bytecode can be generated is a simplified form of C [1]

The language supported by the ClamAV bytecode compiler is a restricted set of C99 with some GNU extensions.

4.1. Differences from C99 and GNU C

These restrictions are enforced at compile time:

- No standard include files. [2]
- The ClamAV API header files are preincluded.
- No external function calls, except to the ClamAV API [3]
- No inline assembly [4]
- Globals can only be readonly constants [5]
- `inline` is C99 inline (equivalent to GNU C89 extern inline), thus it cannot be used outside of the definition of the ClamAV API, you should use `static inline`
- `sizeof(int) == 4` always
- `sizeof(long) == sizeof(long long) == 8` always
- `ptr_diff_t = int`, `intptr_t = int`, `intmax_t = long`, `uintmax_t = unsigned long` [6]
- No pointer to integer casts and integer to pointer casts (pointer arithmetic is allowed though)
- No `__thread` support
- Size of memory region associated with each pointer must be known in each function, thus if you pass a pointer to a function, you must also pass its allocated size as a parameter.
- Endianness must be handled via the `__is_bigendian()` API function call, or via the `cli_{read,write}int{16,32}` wrappers, and not by casting pointers
- Predefines `__CLAMBC__`
- All integer types have fixed width

[1] In the future more languages could be supported, see the Internals Manual on language frontends
[2] For portability reasons: preprocessed C code is not portable
[3] For safety reasons we can't allow the bytecode to call arbitrary system functions
[4] This is both for safety and portability reasons
[5] For thread safety reasons
[6] Note that a pointer's sizeof is runtime-platform dependent, although at compile time sizeof(void*) == 4, at runtime it can be something else. Thus you should avoid using sizeof(pointer)

- `main` or `entrypoint` must have the following prototype: `int main(void)`, the prototype `int main(int argc, char` is not accepted

They are meant to ensure the following:

- Thread safe execution of multiple different bytecodes, and multiple instances of the same bytecode
- Portability to multiple CPU architectures and OSes: the bytecode must execute on both the libclamav/LLVM JIT where that is supported (x86, x86_64, ppc, arm?), and on the libclamav interpreter where that is not supported.
- No external runtime dependency: libclamav should have everything needed to run the bytecode, thus no external calls are allowed, not even to libc!
- Same behaviour on all platforms: fixed size integers.

These restrictions are checked at runtime (checks are inserted at compile time):

- Accessing an out-of-bounds pointer will result in a call to `abort()`
- Calling `abort()` interrupts the execution of the bytecode in a thread safe manner, and doesn't halt ClamAV [1].

The ClamAV API header has further restriction, see the Internals manual.

Although the bytecode undergoes a series of automated tests (see Publishing chapter in Internals manual), the above restrictions don't guarantee that the resulting bytecode will execute correctly! You must still test the code yourself, these restrictions only avoid the most common errors. Although the compiler and verifier aims to accept only code that won't crash ClamAV, no code is 100% perfect, and a bug in the verifier could allow unsafe code be executed by ClamAV.

4.2. Limitations

The bytecode format has the following limitations:

- At most 64k bytecode kinds (hooks)
- At most 64k types (including pointers, and all nested types)
- At most 16 parameters to functions, no vararg functions
- At most 64-bit integers
- No vector types or vector operations
- No opaque types
- No floating point
- Global variable initializer must be compile-time computable
- At most 32k global variables (and at most 32k API globals)
- Pointer indexing at most 15 levels deep (can be worked around if needed by using temporaries)
- No struct return or byval parameters
- At most 32k instructions in a single function
- No Variable Length Arrays

[1] in fact it calls a ClamAV API function, and not the libc abort function.

4.3. Logical signatures

Logical signatures can be used as triggers for executing a bytecode. Instead of describing a logical signatures as a `.ldb` pattern, you use C code which is then translated to a `.ldb`-style logical signature.

Logical signatures in ClamAV support the following operations:

- Sum the count of logical subsignatures that matched inside a subexpression
- Sum the number of different subsignatures that matched inside a subexpression
- Compare the above counts using the $>, =, <$ relation operators
- Perform logical &&, || operations on above boolean values
- Nest subexpressions
- Maximum 64 subexpressions

Out of the above operations the ClamAV Bytecode Compiler doesn't support computing sums of nested subexpressions, (it does support nesting though).

The C code that can be converted into a logical signature must obey these restrictions:

- a function named `logical_trigger` with the following prototype: `bool logical_trigger(void)`
- no function calls, except for `count_match` and `matches`
- no global variable access (except as done by the above 2 functions internally)
- return true when signature should trigger, false otherwise
- use only integer compare instructions, branches, integer *add*, logical *and*, logical *or*, logical *xor*, zero extension, store/load from local variables
- the final boolean expression must be convertible to disjunctive normal form without negation
- the final logical expression must not have more than 64 subexpressions
- it can have early returns (all true returns are unified using ||)
- you can freely use comments, they are ignored
- the final boolean expression cannot be a `true` or `false` constant

The compiler does the following transformations (not necessarily in this order):

- convert shortcircuit boolean operations into non-shortcircuit ones (since all operands are boolean expressions or local variables, it is safe to execute these unconditionally)
- propagate constants
- simplify control flow graph
- (sparse) conditional constant propagation
- dead store elimination
- dead code elimination
- instruction combining (arithmetic simplifications)
- jump threading

If after this transformation the program meets the requirements outlined above, then it is converted to a logical signature. The resulting logical signature is simplified using basic properties of boolean operations, such as associativity, distributivity, De Morgan's law.

The final logical signature is not unique (there might be another logical signature with identical behavior), however the boolean part is in a canonical form: it is in disjunctive normal form, with operands sorted in ascending order.

For best results the C code should consist of:

- local variables declaring the sums you want to use
- a series of `if` branches that `return true`, where the `if`'s condition is a single comparison or a logical *and* of comparisons
- a final `return false`

You can use || in the `if` condition too, but be careful that after expanding to disjunctive normal form, the number of subexpressions doesn't exceed 64.

Note that you do not have to use all the subsignatures you declared in `logical_trigger`, you can do more complicated checks (that wouldn't obey the above restrictions) in the bytecode itself at runtime. The `logical_trigger` function is fully compiled into a logical signature, it won't be a runtime executed function (hence the restrictions).

4.4. Headers and runtime environment

When compiling a bytecode program, `bytecode.h` is automatically included, so you don't need to explicitly include it. These headers (and the compiler itself) predefine certain macros, see Appendix **??** for a full list. In addition the following types are defined:

```
typedef unsigned char uint8_t;
typedef char int8_t;
typedef unsigned short uint16_t;
typedef short int16_t;
typedef unsigned int uint32_t;
typedef int int32_t;
typedef unsigned long uint64_t;
typedef long int64_t;
typedef unsigned int size_t;
typedef int off_t;
typedef struct signature { unsigned id } __Signature;
```

As described in Section 4.1 the width of integer types are fixed, the above typedefs show that.

A bytecode's entrypoint is the function `entrypoint` and it's required by ClamAV to load the bytecode.

Bytecode that is triggered by a logical signature must have a list of virusnames and patterns defined. Bytecodes triggered via hooks can optionally have them, but for example a PE unpacker doesn't need virus names as it only processes the data.

CHAPTER 5

Bytecode security & portability

CHAPTER 6

Reporting bugs

CHAPTER 7
Copyright and License

7.1. The ClamAV Bytecode Compiler

The ClamAV Bytecode Compiler is released under the GNU General Public License version 2.

The following directories are under the GNU General Public License version 2: ClamBC, docs, driver, editor, examples, ifacegen.

```
Copyright (C) 2009 Sourcefire, Inc.

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License version 2 as
published by the Free Software Foundation.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
MA 02110-1301, USA.
```

It uses the LLVM compiler framework, contained in the following directories: llvm, clang. They have this copyright:

```
==============================================================================
LLVM Release License
==============================================================================
University of Illinois/NCSA
Open Source License

Copyright (c) 2003-2009 University of Illinois at Urbana-Champaign.
All rights reserved.

Developed by:

    LLVM Team

    University of Illinois at Urbana-Champaign

    http://llvm.org

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal with
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so, subject to the following conditions:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimers.

    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimers in the
      documentation and/or other materials provided with the distribution.
```

```
    * Neither the names of the LLVM Team, University of Illinois at
      Urbana-Champaign, nor the names of its contributors may be used to
      endorse or promote products derived from this Software without specific
      prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE
SOFTWARE.

==============================================================================
Copyrights and Licenses for Third Party Software Distributed with LLVM:
==============================================================================
The LLVM software contains code written by third parties.  Such software will
have its own individual LICENSE.TXT file in the directory in which it appears.
This file will describe the copyrights, license, and restrictions which apply
to that code.

The disclaimer of warranty in the University of Illinois Open Source License
applies to all code in the LLVM Distribution, and nothing in any of the
other licenses gives permission to use the names of the LLVM Team or the
University of Illinois to endorse or promote products derived from this
Software.

The following pieces of software have additional or alternate copyrights,
licenses, and/or restrictions:

Program             Directory
-------             ---------
Autoconf            llvm/autoconf
                    llvm/projects/ModuleMaker/autoconf
                    llvm/projects/sample/autoconf
CellSPU backend     llvm/lib/Target/CellSPU/README.txt
Google Test         llvm/utils/unittest/googletest
OpenBSD regex       llvm/lib/Support/{reg*, COPYRIGHT.regex}
```

It also uses re2c, contained in driver/clamdriver/re2c. This code is public domain:

```
Originally written by Peter Bumbulis (peter@csg.uwaterloo.ca)

Currently maintained by:
  * Dan Nuffer <nuffer@users.sourceforge.net>
  * Marcus Boerger <helly@users.sourceforge.net>
  * Hartmut Kaiser <hkaiser@users.sourceforge.net>

The re2c distribution can be found at:

    http://sourceforge.net/projects/re2c/

re2c is distributed with no warranty whatever.  The code is certain to
contain errors.  Neither the author nor any contributor takes
responsibility for any consequences of its use.

re2c is in the public domain.  The data structures and algorithms used
in re2c are all either taken from documents available to the general
public or are inventions of the author.  Programs generated by re2c may
be distributed freely.  re2c itself may be distributed freely, in source
or binary, unchanged or modified.  Distributors may charge whatever fees
they can obtain for re2c.

If you do make use of re2c, or incorporate it into a larger project an
acknowledgement somewhere (documentation, research report, etc.) would
be appreciated.
```

7.2. Bytecode

The headers used when compiling bytecode have these license (`clang/lib/Headers/{bcfeatures,bytecode*}.h`):

The other header files in `clang/lib/Headers/` are from clang with this license (see individual files for copyright owner):

When using the ClamAV bytecode compiler to compile your own bytecode programs, you can release it under the license of your choice, provided that you comply with the license of the above header files.