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] 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

1.2. Obtaining the ClamAV Bytecode Compiler

You can obtain the source code in one of the following ways [1]

- 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`

1.3. 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 [2]:

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

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

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

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:

   ```
   $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.).

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

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.

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

```
1 /* Prefix, used for duplicate detection and fixing */
VIRUSNAME_PREFIX("Trojan.Foo")
3 /* You are only allowed to set these virusnames as found */
VIRUSNAMES("A", "B")
5 /* 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

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.

2.2.5. Multiple subsignatures

An example for this is shown in Program 5. Here you see the following new features used: [2]

[1] meaning that subexpression 0 must match

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

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;
}
```

- 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.

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;
}
```

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 [1].

Program 6 Simple regular expression example

```
VIRUSNAME_PREFIX("BC")
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);
                        foundVirus();
      }
      ANY { continue; }
      */
  }
  return 0;
}
```

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.

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

`for(;;) { REGEX_LOOP_BEGIN` this creates the regular expression matcher main loop. It takes the current file byte-by-byte [1] 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.

- `"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.

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

`}` 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.

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

[2] Currently -O0 doesn't work

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:

```
$ 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 libclam
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

You can use `debug_print_str` and `debug_print_int` API calls to print debug messages during the execution of the bytecode.

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

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`

[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

- `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` [1]
- 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
- `main` or `entrypoint` must have the following prototype: `int main(void)`, the prototype `int main(int argc, char *argv[])` 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):

[1] 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)

- 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 A 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

Bytecode API

7.1. Structure types

7.1.1. cli_exe_info Struct Reference

Public Member Functions

- struct cli_exe_section ∗section EBOUNDS (nsections)

Data Fields

- uint32_t offset
- uint32_t ep
- uint16_t nsections
- struct cli_hashset ∗ vinfo

7.1.1.1. Detailed Description

Executable file information

7.1.1.2. Member Function Documentation

7.1.1.2.1. struct cli_exe_section∗ section EBOUNDS (nsections) [read]
Information about all the sections of this file. This array has `nsection` elements

7.1.1.3. Field Documentation

7.1.1.3.1. uint32_t ep Entrypoint of executable

7.1.1.3.2. uint16_t nsections Number of sections

7.1.1.3.3. uint32_t offset Offset where this executable start in file (nonzero if embedded)

7.1.1.3.4. struct cli_hashset* vinfo Hashset for versioninfo matching

7.1.2. cli_exe_section Struct Reference

Data Fields

- uint32_t rva
- uint32_t vsz
- uint32_t raw
- uint32_t rsz
- uint32_t chr
- uint32_t urva
- uint32_t uvsz
- uint32_t uraw
- uint32_t ursz

7.1.2.1. Detailed Description

Section of executable file

7.1.2.2. Field Documentation

7.1.2.2.1. uint32_t chr Section characteristics

7.1.2.2.2. uint32_t raw Raw offset (in file)

7.1.2.2.3. uint32_t rsz Raw size (in file)

7.1.2.2.4. uint32_t rva Relative VirtualAddress

7.1.2.2.5. uint32_t uraw PE - unaligned PointerToRawData

7.1.2.2.6. uint32_t ursz PE - unaligned SizeOfRawData

7.1.2.2.7. uint32_t urva PE - unaligned VirtualAddress

7.1.2.2.8. uint32_t uvsz PE - unaligned VirtualSize

7.1.2.2.9. uint32_t vsz VirtualSize

7.1.3. cli_pe_hook_data Struct Reference

Data Fields

- uint32_t ep
- uint16_t nsections
- struct pe_image_file_hdr file_hdr
- struct pe_image_optional_hdr32 opt32
- struct pe_image_optional_hdr64 opt64
- struct pe_image_data_dir dirs [16]
- uint32_t e_lfanew
- uint32_t overlays
- int32_t overlays_sz
- uint32_t hdr_size

7.1.3.1. Detailed Description

Data for the bytecode PE hook

7.1.3.2. Field Documentation

7.1.3.2.1. struct pe_image_data_dir dirs[16] PE data directory header

7.1.3.2.2. uint32_t e_lfanew address of new exe header

7.1.3.2.3. uint32_t ep EntryPoint as file offset

7.1.3.2.4. struct pe_image_file_hdr file_hdr Header for this PE file

7.1.3.2.5. uint32_t hdr_size internally needed by rawaddr

7.1.3.2.6. uint16_t nsections Number of sections

7.1.3.2.7. struct pe_image_optional_hdr32 opt32 32-bit PE optional header

7.1.3.2.8. struct pe_image_optional_hdr64 opt64 64-bit PE optional header

7.1.3.2.9. uint32_t overlays number of overlays

7.1.3.2.10. int32_t overlays_sz size of overlays

7.1.4. DIS_arg Struct Reference

Data Fields

- enum DIS_ACCESS access_type
- enum DIS_SIZE access_size
- struct DIS_mem_arg mem
- enum X86REGS reg
- uint64_t other

7.1.4.1. Detailed Description

disassembled operand

7.1.4.2. Field Documentation

7.1.4.2.1. enum DIS_SIZE access_size size of access

7.1.4.2.2. enum DIS_ACCESS access_type type of access

7.1.4.2.3. struct DIS_mem_arg mem memory operand

7.1.4.2.4. uint64_t other other operand

7.1.4.2.5. enum X86REGS reg register operand

7.1.5. DIS_fixed Struct Reference

Data Fields

- enum X86OPS x86_opcode
- enum DIS_SIZE operation_size
- enum DIS_SIZE address_size
- uint8_t segment

7.1.5.1. Detailed Description

disassembled instruction

7.1.5.2. Field Documentation

7.1.5.2.1. enum DIS_SIZE address_size size of address

7.1.5.2.2. enum DIS_SIZE operation_size size of operation

7.1.5.2.3. uint8_t segment segment

7.1.5.2.4. enum X86OPS x86_opcode opcode of X86 instruction

7.1.6. DIS_mem_arg Struct Reference

Data Fields

- enum DIS_SIZE access_size
- enum X86REGS scale_reg
- enum X86REGS add_reg
- uint8_t scale
- int32_t displacement

7.1.6.1. Detailed Description

disassembled memory operand: scale_reg∗scale + add_reg + displacement

7.1.6.2. Field Documentation

7.1.6.2.1. enum DIS_SIZE access_size size of access

7.1.6.2.2. enum X86REGS add_reg register used as displacemenet

7.1.6.2.3. int32_t displacement displacement as immediate number

7.1.6.2.4. uint8_t scale scale as immediate number

7.1.6.2.5. enum X86REGS scale_reg register used as scale

7.1.7. DISASM_RESULT Struct Reference

7.1.7.1. Detailed Description

disassembly result, 64-byte, matched by type-8 signatures

7.1.8. pe_image_data_dir Struct Reference

7.1.8.1. Detailed Description

PE data directory header

7.1.9. pe_image_file_hdr Struct Reference

Data Fields

- uint32_t Magic
- uint16_t Machine
- uint16_t NumberOfSections
- uint32_t TimeDateStamp
- uint32_t PointerToSymbolTable
- uint32_t NumberOfSymbols
- uint16_t SizeOfOptionalHeader

7.1.9.1. Detailed Description

Header for this PE file

7.1.9.2. Field Documentation

7.1.9.2.1. uint16_t Machine CPU this executable runs on, see libclamav/pe.c for possible values

7.1.9.2.2. uint32_t Magic PE magic header: PE\0\0

7.1.9.2.3. uint16_t NumberOfSections Number of sections in this executable

7.1.9.2.4. uint32_t NumberOfSymbols debug

7.1.9.2.5. uint32_t PointerToSymbolTable debug

7.1.9.2.6. uint16_t SizeOfOptionalHeader == 224

7.1.9.2.7. uint32_t TimeDateStamp Unreliable

7.1.10. pe_image_optional_hdr32 Struct Reference

Data Fields

- uint8_t MajorLinkerVersion
- uint8_t MinorLinkerVersion
- uint32_t SizeOfCode
- uint32_t SizeOfInitializedData
- uint32_t SizeOfUninitializedData
- uint32_t ImageBase
- uint32_t SectionAlignment
- uint32_t FileAlignment
- uint16_t MajorOperatingSystemVersion
- uint16_t MinorOperatingSystemVersion
- uint16_t MajorImageVersion
- uint16_t MinorImageVersion
- uint32_t CheckSum
- uint32_t NumberOfRvaAndSizes

7.1.10.1. Detailed Description

32-bit PE optional header

7.1.10.2. Field Documentation

7.1.10.2.1. uint32_t CheckSum NT drivers only

7.1.10.2.2. uint32_t FileAlignment usually 32 or 512

7.1.10.2.3. uint32_t ImageBase multiple of 64 KB

7.1.10.2.4. uint16_t MajorImageVersion unreliable

7.1.10.2.5. uint8_t MajorLinkerVersion unreliable

7.1.10.2.6. uint16_t MajorOperatingSystemVersion not used

7.1.10.2.7. uint16_t MinorImageVersion unreliable

7.1.10.2.8. uint8_t MinorLinkerVersion unreliable

7.1.10.2.9. uint16_t MinorOperatingSystemVersion not used

7.1.10.2.10. uint32_t NumberOfRvaAndSizes unreliable

7.1.10.2.11. uint32_t SectionAlignment usually 32 or 4096

7.1.10.2.12. uint32_t SizeOfCode unreliable

7.1.10.2.13. uint32_t SizeOfInitializedData unreliable

7.1.10.2.14. uint32_t SizeOfUninitializedData unreliable

7.1.11. pe_image_optional_hdr64 Struct Reference

Data Fields

- uint8_t MajorLinkerVersion
- uint8_t MinorLinkerVersion
- uint32_t SizeOfCode
- uint32_t SizeOfInitializedData
- uint32_t SizeOfUninitializedData
- uint64_t ImageBase
- uint32_t SectionAlignment
- uint32_t FileAlignment
- uint16_t MajorOperatingSystemVersion
- uint16_t MinorOperatingSystemVersion
- uint16_t MajorImageVersion
- uint16_t MinorImageVersion
- uint32_t CheckSum
- uint32_t NumberOfRvaAndSizes

7.1.11.1. Detailed Description

PE 64-bit optional header

7.1.11.2. Field Documentation

7.1.11.2.1. uint32_t CheckSum NT drivers only

7.1.11.2.2. uint32_t FileAlignment usually 32 or 512

7.1.11.2.3. uint64_t ImageBase multiple of 64 KB

7.1.11.2.4. uint16_t MajorImageVersion unreliable

7.1.11.2.5. uint8_t MajorLinkerVersion unreliable

7.1.11.2.6. uint16_t MajorOperatingSystemVersion not used

7.1.11.2.7. uint16_t MinorImageVersion unreliable

7.1.11.2.8. uint8_t MinorLinkerVersion unreliable

7.1.11.2.9. uint16_t MinorOperatingSystemVersion not used

7.1.11.2.10. uint32_t NumberOfRvaAndSizes unreliable

7.1.11.2.11. uint32_t SectionAlignment usually 32 or 4096

7.1.11.2.12. uint32_t SizeOfCode unreliable

7.1.11.2.13. uint32_t SizeOfInitializedData unreliable

7.1.11.2.14. uint32_t SizeOfUninitializedData unreliable

7.1.12. pe_image_section_hdr Struct Reference

Data Fields

- uint8_t Name [8]
- uint32_t SizeOfRawData
- uint32_t PointerToRawData
- uint32_t PointerToRelocations
- uint32_t PointerToLinenumbers
- uint16_t NumberOfRelocations
- uint16_t NumberOfLinenumbers

7.1.12.1. Detailed Description

PE section header

7.1.12.2. Field Documentation

7.1.12.2.1. uint8_t Name[8] may not end with NULL

7.1.12.2.2. uint16_t NumberOfLinenumbers object files only

7.1.12.2.3. uint16_t NumberOfRelocations object files only

7.1.12.2.4. uint32_t PointerToLinenumbers object files only

7.1.12.2.5. uint32_t PointerToRawData offset to the section's data

7.1.12.2.6. uint32_t PointerToRelocations object files only

7.1.12.2.7. uint32_t SizeOfRawData multiple of FileAlignment

7.2. Low level API

7.2.1. bytecode_api.h File Reference

Enumerations

- enum BytecodeKind { BC_GENERIC = 0 , BC_LOGICAL = 256, BC_PE_UNPACKER }
- enum { SEEK_SET = 0, SEEK_CUR, SEEK_END }

Functions

- uint32_t test1 (uint32_t a, uint32_t b)
- int32_t read (uint8_t *data, int32_t size)

  Reads specified amount of bytes from the current file into a buffer. Also moves current position in the file.

- int32_t write (uint8_t *data, int32_t size)

  Writes the specified amount of bytes from a buffer to the current temporary file.

- int32_t seek (int32_t pos, uint32_t whence)

  Changes the current file position to the specified one.

- uint32_t setvirusname (const uint8_t *name, uint32_t len)
- uint32_t debug_print_str (const uint8_t *str, uint32_t len)
- uint32_t debug_print_uint (uint32_t a)
- uint32_t disasm_x86 (struct DISASM_RESULT *result, uint32_t len)
- uint32_t pe_rawaddr (uint32_t rva)
- int32_t file_find (const uint8_t *data, uint32_t len)
- int32_t file_byteat (uint32_t offset)
- void * malloc (uint32_t size)
- uint32_t test2 (uint32_t a)
- int32_t get_pe_section (struct cli_exe_section *section, uint32_t num)
- int32_t fill_buffer (uint8_t *buffer, uint32_t len, uint32_t filled, uint32_t cursor, uint32_t fill)
- int32_t extract_new (int32_t id)
- int32_t read_number (uint32_t radix)
- int32_t hashset_new (void)

- int32_t hashset_add (int32_t hs, uint32_t key)
- int32_t hashset_remove (int32_t hs, uint32_t key)
- int32_t hashset_contains (int32_t hs, uint32_t key)
- int32_t hashset_done (int32_t id)
- int32_t hashset_empty (int32_t id)
- int32_t buffer_pipe_new (uint32_t size)
- int32_t buffer_pipe_new_fromfile (uint32_t pos)
- uint32_t buffer_pipe_read_avail (int32_t id)
- uint8_t * buffer_pipe_read_get (int32_t id, uint32_t amount)
- int32_t buffer_pipe_read_stopped (int32_t id, uint32_t amount)
- uint32_t buffer_pipe_write_avail (int32_t id)
- uint8_t * buffer_pipe_write_get (int32_t id, uint32_t size)
- int32_t buffer_pipe_write_stopped (int32_t id, uint32_t amount)
- int32_t buffer_pipe_done (int32_t id)
- int32_t inflate_init (int32_t from_buffer, int32_t to_buffer, int32_t window-Bits)
- int32_t inflate_process (int32_t id)
- int32_t inflate_done (int32_t id)
- int32_t bytecode_rt_error (int32_t locationid)
- int32_t jsnorm_init (int32_t from_buffer)
- int32_t jsnorm_process (int32_t id)
- int32_t jsnorm_done (int32_t id)

Variables

- const uint32_t __clambc_match_counts [64]

  Logical signature match counts.

- struct cli_pe_hook_data __clambc_pedata
- const uint32_t __clambc_filesize [1]
- const uint16_t __clambc_kind

7.2.1.1. Detailed Description

7.2.1.2. Enumeration Type Documentation

7.2.1.2.1. anonymous enum

Enumerator:

SEEK_SET set file position to specified absolute position

SEEK_CUR set file position relative to current position

SEEK_END set file position relative to file end

7.2.1.2.2. enum BytecodeKind Bytecode trigger kind

Enumerator:

BC_GENERIC generic bytecode, not tied a specific hook

BC_LOGICAL triggered by a logical signature

BC_PE_UNPACKER a PE unpacker

7.2.1.3. Function Documentation

7.2.1.3.1. int32_t buffer_pipe_done (int32_t *id*) Deallocate memory used by buffer. After this all attempts to use this buffer will result in error. All buffer_pipes are automatically deallocated when bytecode finishes execution.

Parameters:

id ID of buffer_pipe

Returns:

0 on success

7.2.1.3.2. int32_t buffer_pipe_new (uint32_t *size*) Creates a new pipe with the specified buffer size

Parameters:

size size of buffer

Returns:

ID of newly created buffer_pipe

7.2.1.3.3. int32_t buffer_pipe_new_fromfile (uint32_t *pos*) Same as buffer_pipe_new, except the pipe's input is tied to the current file, at the specified position.

Parameters:

pos starting position of pipe input in current file

Returns:

ID of newly created buffer_pipe

7.2.1.3.4. uint32_t buffer_pipe_read_avail (int32_t *id*) Returns the amount of bytes available to read.

Parameters:

id ID of buffer_pipe

Returns:

amount of bytes available to read

7.2.1.3.5. uint8_t* buffer_pipe_read_get (int32_t *id*, uint32_t *amount*) Returns a pointer to the buffer for reading. The 'amount' parameter should be obtained by a call to buffer_pipe_read_avail().

Parameters:

id ID of buffer_pipe

amount to read

Returns:

pointer to buffer, or NULL if buffer has less than specified amount

7.2.1.3.6. int32_t buffer_pipe_read_stopped (int32_t *id*, uint32_t *amount*) Updates read cursor in buffer_pipe.

Parameters:

id ID of buffer_pipe

amount amount of bytes to move read cursor

Returns:

0 on success

7.2.1.3.7. uint32_t buffer_pipe_write_avail (int32_t *id*) Returns the amount of bytes available for writing.

Parameters:

id ID of buffer_pipe

Returns:

amount of bytes available for writing

7.2.1.3.8. uint8_t* buffer_pipe_write_get (int32_t *id*, uint32_t *size*) Returns pointer to writable buffer. The 'amount' parameter should be obtained by a call to buffer_pipe_write_avail().

Parameters:

id ID of buffer_pipe

size amount of bytes to write

Returns:

pointer to write buffer, or NULL if requested amount is more than what is available in the buffer

7.2.1.3.9. int32_t buffer_pipe_write_stopped (int32_t *id*, uint32_t *amount*) Updates the write cursor in buffer_pipe.

Parameters:

id ID of buffer_pipe

amount amount of bytes to move write cursor

Returns:

0 on success

7.2.1.3.10. int32_t bytecode_rt_error (int32_t *locationid*) Report a runtime error at the specified locationID.

Parameters:

locationid (line << 8) | (column&0xff)

Returns:

0

7.2.1.3.11. uint32_t debug_print_str (const uint8_t * *str*, uint32_t *len*) Prints a debug message.

Parameters:

← ***str*** Message to print

← ***len*** length of message to print

Returns:

0

7.2.1.3.12. uint32_t debug_print_uint (uint32_t *a*) Prints a number as a debug message.

Parameters:

← ***a*** number to print

Returns:

0

7.2.1.3.13. uint32_t disasm_x86 (struct DISASM_RESULT * *result*, uint32_t *len*) Disassembles starting from current file position, the specified amount of bytes.

Parameters:

→ ***result*** pointer to struct holding result

← ***len*** how many bytes to disassemble

Returns:

0 for success

You can use lseek to disassemble starting from a different location. This is a low-level API, the result is in ClamAV type-8 signature format (64 bytes/instruction).

See also:

DisassembleAt

7.2.1.3.14. int32_t extract_new (int32_t *id*) Prepares for extracting a new file, if we've already extracted one it scans it.

Parameters:

← ***id*** an id for the new file (for example position in container)

Returns:

1 if previous extracted file was infected

7.2.1.3.15. int32_t file_byteat (uint32_t *offset*) Read a single byte from current file

Parameters:

offset file offset

Returns:

byte at offset `off` in the current file, or -1 if offset is invalid

7.2.1.3.16. int32_t file_find (const uint8_t ∗ *data*, uint32_t *len*) Looks for the specified sequence of bytes in the current file.

Parameters:

← ***data*** the sequence of bytes to look for

len length of `data`, cannot be more than 1024

Returns:

offset in the current file if match is found, -1 otherwise

7.2.1.3.17. int32_t fill_buffer (uint8_t ∗ *buffer*, uint32_t *len*, uint32_t *filled*, uint32_t *cursor*, uint32_t *fill*) Fills the specified buffer with at least `fill` bytes.

Parameters:

→ ***buffer*** the buffer to fill

← ***len*** length of buffer

← ***filled*** how much of the buffer is currently filled

← ***cursor*** position of cursor in buffer

← ***fill*** amount of bytes to fill in (0 is valid)

Returns:

<0 on error, 0 on EOF, number bytes available in buffer (starting from 0) The character at the cursor will be at position 0 after this call.

7.2.1.3.18. int32_t get_pe_section (struct cli_exe_section * *section*, uint32_t *num*) Gets information about the specified PE section.

Parameters:

→ ***section*** PE section information will be stored here

← ***num*** PE section number

7.2.1.3.19. int32_t hashset_add (int32_t *hs*, uint32_t *key*) Add a new 32-bit key to the hashset.

Parameters:

hs ID of hashset (from hashset_new)

key the key to add

Returns:

0 on success

7.2.1.3.20. int32_t hashset_contains (int32_t *hs*, uint32_t *key*) Returns whether the hashset contains the specified key.

Parameters:

hs ID of hashset (from hashset_new)

key the key to lookup

Returns:

1 if found, 0 if not found, <0 on invalid hashset ID

7.2.1.3.21. int32_t hashset_done (int32_t *id*) Deallocates the memory used by the specified hashset. Trying to use the hashset after this will result in an error. The hashset may not be used after this. All hashsets are automatically deallocated when bytecode finishes execution.

Parameters:

id ID of hashset (from hashset_new)

Returns:

0 on success

7.2.1.3.22. int32_t hashset_empty (int32_t *id*) Returns whether the hashset is empty.

Parameters:

id of hashset (from hashset_new)

Returns:

0 on success

7.2.1.3.23. int32_t hashset_new (void) Creates a new hashset and returns its id.

Returns:

ID for new hashset

7.2.1.3.24. int32_t hashset_remove (int32_t *hs*, uint32_t *key*) Remove a 32-bit key from the hashset.

Parameters:

hs ID of hashset (from hashset_new)

key the key to add

Returns:

0 on success

7.2.1.3.25. int32_t inflate_done (int32_t *id*) Deallocates inflate data structure. Using the inflate data structure after this will result in an error. All inflate data structures are automatically deallocated when bytecode finishes execution.

Parameters:

id ID of inflate data structure

Returns:

0 on success.

7.2.1.3.26. int32_t inflate_init (int32_t *from_buffer*, int32_t *to_buffer*, int32_t *windowBits*) Initializes inflate data structures for decompressing data 'from_buffer' and writing uncompressed uncompressed data 'to_buffer'.

Parameters:

from_buffer ID of buffer_pipe to read compressed data from

to_buffer ID of buffer_pipe to write decompressed data to

windowBits (see zlib documentation)

Returns:

ID of newly created inflate data structure, <0 on failure

7.2.1.3.27. int32_t inflate_process (int32_t *id*) Inflate all available data in the input buffer, and write to output buffer. Stops when the input buffer becomes empty, or write buffer becomes full. Also attempts to recover from corrupted inflate stream (via inflateSync). This function can be called repeatedly on success after filling the input buffer, and flushing the output buffer. The inflate stream is done processing when 0 bytes are available from output buffer, and input buffer is not empty.

Parameters:

id ID of inflate data structure

Returns:

0 on success, zlib error code otherwise

7.2.1.3.28. int32_t jsnorm_done (int32_t *id*) Flushes JS normalizer.

Parameters:

id ID of js normalizer to flush

7.2.1.3.29. int32_t jsnorm_init (int32_t *from_buffer*) Initializes JS normalizer for reading 'from_buffer'. Normalized JS will be written to a single tempfile, one normalized JS per line, and automatically scanned when the bytecode finishes execution.

Parameters:

from_buffer ID of buffer_pipe to read javascript from

Returns:

ID of JS normalizer, <0 on failure

7.2.1.3.30. int32_t jsnorm_process (int32_t *id*) Normalize all javascript from the input buffer, and write to tempfile. You can call this function repeatedly on success, if you (re)fill the input buffer.

Parameters:

id ID of JS normalizer

Returns:

0 on success, <0 on failure

7.2.1.3.31. void$*$ malloc (uint32_t *size*) Allocates memory. Currently this memory is freed automatically on exit from the bytecode, and there is no way to free it sooner.

Parameters:

size amount of memory to allocate in bytes

Returns:

pointer to allocated memory

7.2.1.3.32. uint32_t pe_rawaddr (uint32_t *rva*) Converts a RVA (Relative Virtual Address) to an absolute PE file offset.

Parameters:

rva a rva address from the PE file

Returns:

absolute file offset mapped to the `rva`, or PE_INVALID_RVA if the `rva` is invalid.

7.2.1.3.33. int32_t read (uint8_t ∗ *data*, int32_t *size*)
Reads specified amount of bytes from the current file into a buffer. Also moves current position in the file.

Parameters:

← ***size*** amount of bytes to read

→ ***data*** pointer to buffer where data is read into

Returns:

amount read.

7.2.1.3.34. int32_t read_number (uint32_t *radix*) Reads a number in the specified radix starting from the current position. Non-numeric characters are ignored.

Parameters:

← ***radix*** 10 or 16

Returns:

the number read

7.2.1.3.35. int32_t seek (int32_t *pos*, uint32_t *whence*)
Changes the current file position to the specified one.

See also:

SEEK_SET, SEEK_CUR, SEEK_END

Parameters:

← ***pos*** offset (absolute or relative depending on `whence` param)

← ***whence*** one of `SEEK_SET`, `SEEK_CUR`, `SEEK_END`

Returns:

absolute position in file

7.2.1.3.36. uint32_t setvirusname (const uint8_t * *name*, uint32_t *len*) Sets the name of the virus found.

Parameters:

← ***name*** the name of the virus

← ***len*** length of the virusname

Returns:

0

7.2.1.3.37. uint32_t test1 (uint32_t *a*, uint32_t *b*) Test api.

Parameters:

a 0xf00dbeef

b 0xbeeff00d

Returns:

0x12345678 if parameters match, 0x55 otherwise

7.2.1.3.38. uint32_t test2 (uint32_t *a*) Test api2.

Parameters:

a 0xf00d

Returns:

0xd00f if parameter matches, 0x5555 otherwise

7.2.1.3.39. int32_t write (uint8_t * *data*, int32_t *size*)

Writes the specified amount of bytes from a buffer to the current temporary file.

Parameters:

← ***data*** pointer to buffer of data to write

← ***size*** amount of bytes to write `size` bytes to temporary file, from the buffer pointed to byte

Returns:

amount of bytes successfully written

7.2.1.4. Variable Documentation

7.2.1.4.1. const uint32_t __clambc_filesize[1]

File size (max 4G)

7.2.1.4.2. const uint16_t __clambc_kind

Kind of the bytecode

7.2.1.4.3. const uint32_t __clambc_match_counts[64]

Logical signature match counts.
This is a low-level variable, use the Macros in bytecode_local.h instead to access it.

7.2.1.4.4. struct cli_pe_hook_data __clambc_pedata

PE data, if this is a PE hook

7.2.2. bytecode_disasm.h File Reference

Data Structures

- struct DISASM_RESULT

Enumerations

- enum X86OPS { ,
  OP_AAA, OP_AAD, OP_AAM, OP_AAS,
  OP_ADD, OP_ADC, OP_AND, OP_ARPL,
  OP_BOUND, OP_BSF, OP_BSR, OP_BSWAP,
  OP_BT, OP_BTC, OP_BTR, OP_BTS,
  OP_CALL, OP_CDQ , OP_CWDE, OP_CBW,

OP_CLC, OP_CLD, OP_CLI, OP_CLTS,
OP_CMC, OP_CMOVO, OP_CMOVNO, OP_CMOVC,
OP_CMOVNC, OP_CMOVZ, OP_CMOVNZ, OP_CMOVBE,
OP_CMOVA, OP_CMOVS, OP_CMOVNS, OP_CMOVP,
OP_CMOVNP, OP_CMOVL, OP_CMOVGE, OP_CMOVLE,
OP_CMOVG, OP_CMP, OP_CMPSD, OP_CMPSW,
OP_CMPSB, OP_CMPXCHG, OP_CMPXCHG8B, OP_CPUID,
OP_DAA, OP_DAS, OP_DEC, OP_DIV,
OP_ENTER, OP_FWAIT, OP_HLT, OP_IDIV,
OP_IMUL, OP_INC, OP_IN, OP_INSD,
OP_INSW, OP_INSB, OP_INT, OP_INT3,
OP_INTO, OP_INVD, OP_INVLPG, OP_IRET,
OP_JO, OP_JNO, OP_JC, OP_JNC,
OP_JZ, OP_JNZ, OP_JBE, OP_JA,
OP_JS, OP_JNS, OP_JP, OP_JNP,
OP_JL, OP_JGE, OP_JLE, OP_JG,
OP_JMP, OP_LAHF, OP_LAR, OP_LDS,
OP_LES, OP_LFS, OP_LGS, OP_LEA,
OP_LEAVE, OP_LGDT, OP_LIDT, OP_LLDT,
OP_PREFIX_LOCK, OP_LODSD, OP_LODSW, OP_LODSB,
OP_LOOP, OP_LOOPE, OP_LOOPNE, OP_JECXZ,
OP_LSL, OP_LSS, OP_LTR, OP_MOV,
OP_MOVSD, OP_MOVSW, OP_MOVSB, OP_MOVSX,
OP_MOVZX, OP_MUL, OP_NEG, OP_NOP,
OP_NOT, OP_OR, OP_OUT, OP_OUTSD,
OP_OUTSW, OP_OUTSB, OP_PUSH, OP_PUSHAD ,
OP_PUSHFD , OP_POP, OP_POPAD, OP_POPFD ,
OP_RCL, OP_RCR, OP_RDMSR, OP_RDPMC,
OP_RDTSC, OP_PREFIX_REPE, OP_PREFIX_REPNE, OP_RETF,
OP_RETN, OP_ROL, OP_ROR, OP_RSM,
OP_SAHF, OP_SAR, OP_SBB, OP_SCASD,

OP_SCASW, OP_SCASB, OP_SETO, OP_SETNO,
OP_SETC, OP_SETNC, OP_SETZ, OP_SETNZ,
OP_SETBE, OP_SETA, OP_SETS, OP_SETNS,
OP_SETP, OP_SETNP, OP_SETL, OP_SETGE,
OP_SETLE, OP_SETG, OP_SGDT, OP_SIDT,
OP_SHL, OP_SHLD, OP_SHR, OP_SHRD,
OP_SLDT, OP_STOSD, OP_STOSW, OP_STOSB,
OP_STR, OP_STC, OP_STD, OP_STI,
OP_SUB, OP_SYSCALL, OP_SYSENTER, OP_SYSEXIT,
OP_SYSRET, OP_TEST, OP_UD2, OP_VERR,
OP_VERRW, OP_WBINVD, OP_WRMSR, OP_XADD,
OP_XCHG, OP_XLAT, OP_XOR , OP_FPU,
OP_F2XM1, OP_FABS, OP_FADD, OP_FADDP,
OP_FBLD, OP_FBSTP, OP_FCHS, OP_FCLEX,
OP_FCMOVB, OP_FCMOVBE, OP_FCMOVE, OP_FCMOVNB,
OP_FCMOVNBE, OP_FCMOVNE, OP_FCMOVNU, OP_FCMOVU,
OP_FCOM, OP_FCOMI, OP_FCOMIP, OP_FCOMP,
OP_FCOMPP, OP_FCOS, OP_FDECSTP, OP_FDIV,
OP_FDIVP, OP_FDIVR, OP_FDIVRP, OP_FFREE,
OP_FIADD, OP_FICOM, OP_FICOMP, OP_FIDIV,
OP_FIDIVR, OP_FILD, OP_FIMUL, OP_FINCSTP,
OP_FINIT, OP_FIST, OP_FISTP, OP_FISTTP,
OP_FISUB, OP_FISUBR, OP_FLD, OP_FLD1,
OP_FLDCW, OP_FLDENV, OP_FLDL2E, OP_FLDL2T,
OP_FLDLG2, OP_FLDLN2, OP_FLDPI, OP_FLDZ,
OP_FMUL, OP_FMULP, OP_FNOP, OP_FPATAN,
OP_FPREM, OP_FPREM1, OP_FPTAN, OP_FRNDINT,
OP_FRSTOR, OP_FSCALE , OP_FSINCOS, OP_FSQRT,
OP_FSAVE, OP_FST, OP_FSTCW, OP_FSTENV,
OP_FSTP, OP_FSTSW, OP_FSUB, OP_FSUBP,
OP_FSUBR, OP_FSUBRP, OP_FTST, OP_FUCOM,

OP_FUCOMI, OP_FUCOMIP, OP_FUCOMP, OP_FUCOMPP,
OP_FXAM, OP_FXCH, OP_FXTRACT, OP_FYL2X,
OP_FYL2XP1 }

- enum DIS_ACCESS {
  ACCESS_NOARG, ACCESS_IMM, ACCESS_REL, ACCESS_REG,
  ACCESS_MEM }
- enum DIS_SIZE {
  SIZEB, SIZEW, SIZED, SIZEF,
  SIZEQ, SIZET, SIZEPTR }
- enum X86REGS

7.2.2.1. Detailed Description

7.2.2.2. Enumeration Type Documentation

7.2.2.2.1. enum DIS_ACCESS Access type

Enumerator:

ACCESS_NOARG arg not present

ACCESS_IMM immediate

ACCESS_REL +/- immediate

ACCESS_REG register

ACCESS_MEM [memory]

7.2.2.2.2. enum DIS_SIZE for mem access, immediate and relative

Enumerator:

SIZEB Byte size access

SIZEW Word size access

SIZED Doubleword size access

SIZEF 6-byte access (seg+reg pair)

SIZEQ Quadword access

SIZET 10-byte access

SIZEPTR ptr

7.2.2.2.3. enum X86OPS

X86 opcode

Enumerator:

OP_AAA Ascii Adjust after Addition

OP_AAD Ascii Adjust AX before Division

OP_AAM Ascii Adjust AX after Multiply

OP_AAS Ascii Adjust AL after Subtraction

OP_ADD Add

OP_ADC Add with Carry

OP_AND Logical And

OP_ARPL Adjust Requested Privilege Level

OP_BOUND Check Array Index Against Bounds

OP_BSF Bit Scan Forward

OP_BSR Bit Scan Reverse

OP_BSWAP Byte Swap

OP_BT Bit Test

OP_BTC Bit Test and Complement

OP_BTR Bit Test and Reset

OP_BTS Bit Test and Set

OP_CALL Call

OP_CDQ Convert DoubleWord to QuadWord

OP_CWDE Convert Word to DoubleWord

OP_CBW Convert Byte to Word

OP_CLC Clear Carry Flag

OP_CLD Clear Direction Flag

OP_CLI Clear Interrupt Flag

OP_CLTS Clear Task-Switched Flag in CR0

OP_CMC Complement Carry Flag

OP_CMOVO Conditional Move if Overflow

OP_CMOVNO Conditional Move if Not Overflow

OP_CMOVC Conditional Move if Carry

OP_CMOVNC Conditional Move if Not Carry

OP_CMOVZ Conditional Move if Zero

OP_CMOVNZ Conditional Move if Non-Zero

OP_CMOVBE Conditional Move if Below or Equal

OP_CMOVA Conditional Move if Above

OP_CMOVS Conditional Move if Sign

OP_CMOVNS Conditional Move if Not Sign

OP_CMOVP Conditional Move if Parity

OP_CMOVNP Conditional Move if Not Parity

OP_CMOVL Conditional Move if Less

OP_CMOVGE Conditional Move if Greater or Equal

OP_CMOVLE Conditional Move if Less than or Equal

OP_CMOVG Conditional Move if Greater

OP_CMP Compare

OP_CMPSD Compare String DoubleWord

OP_CMPSW Compare String Word

OP_CMPSB Compare String Byte

OP_CMPXCHG Compare and Exchange

OP_CMPXCHG8B Compare and Exchange Bytes

OP_CPUID CPU Identification

OP_DAA Decimal Adjust AL after Addition

OP_DAS Decimal Adjust AL after Subtraction

OP_DEC Decrement by 1

OP_DIV Unsigned Divide

OP_ENTER Make Stack Frame for Procedure Parameters

OP_FWAIT Wait

OP_HLT Halt

OP_IDIV Signed Divide

OP_IMUL Signed Multiply

OP_INC Increment by 1

OP_IN INput from port

OP_INSD INput from port to String Doubleword

OP_INSW INput from port to String Word

OP_INSB INput from port to String Byte

OP_INT INTerrupt

OP_INT3 INTerrupt 3 (breakpoint)

OP_INTO INTerrupt 4 if Overflow

OP_INVD Invalidate Internal Caches

OP_INVLPG Invalidate TLB Entry

OP_IRET Interrupt Return

OP_JO Jump if Overflow

OP_JNO Jump if Not Overflow

OP_JC Jump if Carry

OP_JNC Jump if Not Carry

OP_JZ Jump if Zero

OP_JNZ Jump if Not Zero

OP_JBE Jump if Below or Equal

OP_JA Jump if Above

OP_JS Jump if Sign

OP_JNS Jump if Not Sign

OP_JP Jump if Parity

OP_JNP Jump if Not Parity

OP_JL Jump if Less

OP_JGE Jump if Greater or Equal

OP_JLE Jump if Less or Equal

OP_JG Jump if Greater

OP_JMP Jump (unconditional)

OP_LAHF Load Status Flags into AH Register

OP_LAR load Access Rights Byte

OP_LDS Load Far Pointer into DS

OP_LES Load Far Pointer into ES

OP_LFS Load Far Pointer into FS

OP_LGS Load Far Pointer into GS

OP_LEA Load Effective Address

OP_LEAVE High Level Procedure Exit

OP_LGDT Load Global Descript Table Register

OP_LIDT Load Interrupt Descriptor Table Register

OP_LLDT Load Local Descriptor Table Register

OP_PREFIX_LOCK Assert LOCK# Signal Prefix

OP_LODSD Load String Dword

OP_LODSW Load String Word

OP_LODSB Load String Byte

OP_LOOP Loop According to ECX Counter

OP_LOOPE Loop According to ECX Counter and ZF=1

OP_LOOPNE Looop According to ECX Counter and ZF=0

OP_JECXZ Jump if ECX is Zero

OP_LSL Load Segment Limit

OP_LSS Load Far Pointer into SS

OP_LTR Load Task Register

OP_MOV Move

OP_MOVSD Move Data from String to String Doubleword

OP_MOVSW Move Data from String to String Word

OP_MOVSB Move Data from String to String Byte

OP_MOVSX Move with Sign-Extension

OP_MOVZX Move with Zero-Extension

OP_MUL Unsigned Multiply

OP_NEG Two's Complement Negation

OP_NOP No Operation

OP_NOT One's Complement Negation

OP_OR Logical Inclusive OR

OP_OUT Output to Port

OP_OUTSD Output String to Port Doubleword

OP_OUTSW Output String to Port Word

OP_OUTSB Output String to Port Bytes

OP_PUSH Push Onto the Stack

OP_PUSHAD Push All Double General Purpose Registers

OP_PUSHFD Push EFLAGS Register onto the Stack

OP_POP Pop a Value from the Stack

OP_POPAD Pop All Double General Purpose Registers from the Stack

OP_POPFD Pop Stack into EFLAGS Register

OP_RCL Rotate Carry Left

OP_RCR Rotate Carry Right

OP_RDMSR Read from Model Specific Register

OP_RDPMC Read Performance Monitoring Counters

OP_RDTSC Read Time-Stamp Coutner

OP_PREFIX_REPE Repeat String Operation Prefix while Equal

OP_PREFIX_REPNE Repeat String Operation Prefix while Not Equal

OP_RETF Return from Far Procedure

OP_RETN Return from Near Procedure

OP_ROL Rotate Left

OP_ROR Rotate Right

OP_RSM Resumse from System Management Mode

OP_SAHF Store AH into Flags

OP_SAR Shift Arithmetic Right

OP_SBB Subtract with Borrow

OP_SCASD Scan String Doubleword

OP_SCASW Scan String Word

OP_SCASB Scan String Byte

OP_SETO Set Byte on Overflow

OP_SETNO Set Byte on Not Overflow

OP_SETC Set Byte on Carry

OP_SETNC Set Byte on Not Carry

OP_SETZ Set Byte on Zero

OP_SETNZ Set Byte on Not Zero

OP_SETBE Set Byte on Below or Equal

OP_SETA Set Byte on Above

OP_SETS Set Byte on Sign

OP_SETNS Set Byte on Not Sign

OP_SETP Set Byte on Parity

OP_SETNP Set Byte on Not Parity

OP_SETL Set Byte on Less

OP_SETGE Set Byte on Greater or Equal

OP_SETLE Set Byte on Less or Equal

OP_SETG Set Byte on Greater

OP_SGDT Store Global Descriptor Table Register

OP_SIDT Store Interrupt Descriptor Table Register

OP_SHL Shift Left

OP_SHLD Double Precision Shift Left

OP_SHR Shift Right

OP_SHRD Double Precision Shift Right

OP_SLDT Store Local Descriptor Table Register

OP_STOSD Store String Doubleword

OP_STOSW Store String Word

OP_STOSB Store String Byte

OP_STR Store Task Register

OP_STC Set Carry Flag

OP_STD Set Direction Flag

OP_STI Set Interrupt Flag

OP_SUB Subtract

OP_SYSCALL Fast System Call

OP_SYSENTER Fast System Call

OP_SYSEXIT Fast Return from Fast System Call

OP_SYSRET Return from Fast System Call

OP_TEST Logical Compare

OP_UD2 Undefined Instruction

OP_VERR Verify a Segment for Reading

OP_VERRW Verify a Segment for Writing

OP_WBINVD Write Back and Invalidate Cache

OP_WRMSR Write to Model Specific Register

OP_XADD Exchange and Add

OP_XCHG Exchange Register/Memory with Register

OP_XLAT Table Look-up Translation

OP_XOR Logical Exclusive OR

OP_FPU FPU operation

OP_F2XM1 Compute 2x-1

OP_FABS Absolute Value

OP_FADD Floating Point Add

OP_FADDP Floating Point Add, Pop

OP_FBLD Load Binary Coded Decimal

OP_FBSTP Store BCD Integer and Pop

OP_FCHS Change Sign

OP_FCLEX Clear Exceptions

OP_FCMOVB Floating Point Move on Below

OP_FCMOVBE Floating Point Move on Below or Equal

OP_FCMOVE Floating Point Move on Equal

OP_FCMOVNB Floating Point Move on Not Below

OP_FCMOVNBE Floating Point Move on Not Below or Equal

OP_FCMOVNE Floating Point Move on Not Equal

OP_FCMOVNU Floating Point Move on Not Unordered

OP_FCMOVU Floating Point Move on Unordered

OP_FCOM Compare Floating Pointer Values and Set FPU Flags

OP_FCOMI Compare Floating Pointer Values and Set EFLAGS

OP_FCOMIP Compare Floating Pointer Values and Set EFLAGS, Pop

OP_FCOMP Compare Floating Pointer Values and Set FPU Flags, Pop

OP_FCOMPP Compare Floating Pointer Values and Set FPU Flags, Pop Twice

OP_FCOS Cosine

OP_FDECSTP Decrement Stack Top Pointer

OP_FDIV Floating Point Divide

OP_FDIVP Floating Point Divide, Pop

OP_FDIVR Floating Point Reverse Divide

OP_FDIVRP Floating Point Reverse Divide, Pop

OP_FFREE Free Floating Point Register

OP_FIADD Floating Point Add

OP_FICOM Compare Integer

OP_FICOMP Compare Integer, Pop

OP_FIDIV Floating Point Divide by Integer

OP_FIDIVR Floating Point Reverse Divide by Integer

OP_FILD Load Integer

OP_FIMUL Floating Point Multiply with Integer

OP_FINCSTP Increment Stack-Top Pointer

OP_FINIT Initialize Floating-Point Unit

OP_FIST Store Integer

OP_FISTP Store Integer, Pop

OP_FISTTP Store Integer with Truncation

OP_FISUB Floating Point Integer Subtract

OP_FISUBR Floating Point Reverse Integer Subtract

OP_FLD Load Floating Point Value

OP_FLD1 Load Constant 1

OP_FLDCW Load x87 FPU Control Word

OP_FLDENV Load x87 FPU Environment

OP_FLDL2E Load Constant log_2(e)

OP_FLDL2T Load Constant log_2(10)

OP_FLDLG2 Load Constant log_10(2)

OP_FLDLN2 Load Constant log_e(2)

OP_FLDPI Load Constant PI

OP_FLDZ Load Constant Zero

OP_FMUL Floating Point Multiply
OP_FMULP Floating Point Multiply, Pop
OP_FNOP No Operation
OP_FPATAN Partial Arctangent
OP_FPREM Partial Remainder
OP_FPREM1 Partial Remainder
OP_FPTAN Partial Tangent
OP_FRNDINT Round to Integer
OP_FRSTOR Restore x86 FPU State
OP_FSCALE Scale
OP_FSINCOS Sine and Cosine
OP_FSQRT Square Root
OP_FSAVE Store x87 FPU State
OP_FST Store Floating Point Value
OP_FSTCW Store x87 FPU Control Word
OP_FSTENV Store x87 FPU Environment
OP_FSTP Store Floating Point Value, Pop
OP_FSTSW Store x87 FPU Status Word
OP_FSUB Floating Point Subtract
OP_FSUBP Floating Point Subtract, Pop
OP_FSUBR Floating Point Reverse Subtract
OP_FSUBRP Floating Point Reverse Subtract, Pop
OP_FTST Floating Point Test
OP_FUCOM Floating Point Unordered Compare
OP_FUCOMI Floating Point Unordered Compare with Integer
OP_FUCOMIP Floating Point Unorder Compare with Integer, Pop
OP_FUCOMP Floating Point Unorder Compare, Pop
OP_FUCOMPP Floating Point Unorder Compare, Pop Twice
OP_FXAM Examine ModR/M
OP_FXCH Exchange Register Contents
OP_FXTRACT Extract Exponent and Significand
OP_FYL2X Compute y$*$log2x
OP_FYL2XP1 Compute y$*$log2(x+1)

7.2.2.2.4. enum X86REGS

X86 registers

7.2.3. bytecode_execs.h File Reference

Data Structures

- struct cli_exe_section
- struct cli_exe_info

7.2.3.1. Detailed Description

7.2.4. bytecode_pe.h File Reference

Data Structures

- struct pe_image_file_hdr
- struct pe_image_data_dir
- struct pe_image_optional_hdr32
- struct pe_image_optional_hdr64
- struct pe_image_section_hdr
- struct cli_pe_hook_data

7.2.4.1. Detailed Description

7.3. High level API

7.3.1. bytecode_local.h File Reference

Data Structures

- struct DIS_mem_arg
- struct DIS_arg
- struct DIS_fixed

Defines

- #define VIRUSNAME_PREFIX(name) const char __clambc_virusname_prefix[] = name;
- #define VIRUSNAMES(...) const char *const __clambc_virusnames[] = {__VA_ARGS__};

- #define SIGNATURES_DECL_BEGIN struct __Signatures {
- #define DECLARE_SIGNATURE(name)
- #define SIGNATURES_DECL_END };
- #define TARGET(tgt) const unsigned short __Target = (tgt);
- #define SIGNATURES_DEF_BEGIN
- #define DEFINE_SIGNATURE(name, hex)
- #define SIGNATURES_END };\

Functions

- static force_inline uint32_t count_match (__Signature sig)
- static force_inline uint32_t matches (__Signature sig)
- static force_inline void foundVirus (const char *virusname)
- static force_inline uint32_t getFilesize (void)
- bool __is_bigendian (void) __attribute__((const)) __attribute__((nothrow))
- static uint32_t force_inline le32_to_host (uint32_t v)
- static uint32_t force_inline le64_to_host (uint32_t v)
- static uint16_t force_inline le16_to_host (uint16_t v)
- static uint32_t force_inline cli_readint32 (const void *buff)
- static uint16_t force_inline cli_readint16 (const void *buff)
- static void force_inline cli_writeint32 (void *offset, uint32_t v)
- static force_inline bool hasExeInfo (void)
- static force_inline bool isPE64 (void)
- static static force_inline force_inline uint8_t getPEMajorLinkerVersion (void)
- static force_inline uint8_t getPEMinorLinkerVersion (void)
- static force_inline uint32_t getPESizeOfCode (void)
- static force_inline uint32_t getPESizeOfInitializedData (void)
- static force_inline uint32_t getPESizeOfUninitializedData (void)
- static force_inline uint32_t getPEBaseOfCode (void)
- static force_inline uint32_t getPEBaseOfData (void)
- static force_inline uint64_t getPEImageBase (void)
- static force_inline uint32_t getPESectionAlignment (void)
- static force_inline uint32_t getPEFileAlignment (void)
- static force_inline uint16_t getPEMajorOperatingSystemVersion (void)
- static force_inline uint16_t getPEMinorOperatingSystemVersion (void)
- static force_inline uint16_t getPEMajorImageVersion (void)

- static force_inline uint16_t getPEMinorImageVersion (void)
- static force_inline uint16_t getPEMajorSubsystemVersion (void)
- static force_inline uint16_t getPEMinorSubsystemVersion (void)
- static force_inline uint32_t getPEWin32VersionValue (void)
- static force_inline uint32_t getPESizeOfImage (void)
- static force_inline uint32_t getPESizeOfHeaders (void)
- static force_inline uint32_t getPECheckSum (void)
- static force_inline uint16_t getPESubsystem (void)
- static force_inline uint16_t getPEDllCharacteristics (void)
- static force_inline uint32_t getPESizeOfStackReserve (void)
- static force_inline uint32_t getPESizeOfStackCommit (void)
- static force_inline uint32_t getPESizeOfHeapReserve (void)
- static force_inline uint32_t getPESizeOfHeapCommit (void)
- static force_inline uint32_t getPELoaderFlags (void)
- static force_inline uint16_t getPEMachine ()
- static force_inline uint32_t getPETimeDateStamp ()
- static force_inline uint32_t getPEPointerToSymbolTable ()
- static force_inline uint32_t getPENumberOfSymbols ()
- static force_inline uint16_t getPESizeOfOptionalHeader ()
- static force_inline uint16_t getPECharacteristics ()
- static force_inline bool getPEisDLL ()
- static force_inline uint32_t getPEDataDirRVA (unsigned n)
- static force_inline uint32_t getPEDataDirSize (unsigned n)
- static force_inline uint16_t getNumberOfSections (void)
- static uint32_t getPELFANew (void)
- static force_inline int readPESectionName (unsigned char name[8], unsigned n)
- static force_inline uint32_t getEntryPoint (void)
- static force_inline uint32_t getExeOffset (void)
- static force_inline uint32_t getImageBase (void)
- static force_inline bool readRVA (uint32_t rva, void *buf, size_t bufsize)
- static void * memchr (const void *s, int c, size_t n)
- void * memset (void *src, int c, uintptr_t n) __attribute__((nothrow)) __attribute__((__nonnull__((1))))
- void * memmove (void *dst, const void *src, uintptr_t n) __attribute__((__nothrow__)) __attribute__((__nonnull__(1

- void ∗void ∗ memcpy (void ∗restrict dst, const void ∗restrict src, uintptr_t n) __attribute__((__nothrow__)) __attribute__((__nonnull__(1
- void ∗void ∗int memcmp (const void ∗s1, const void ∗s2, uint32_t n) __attribute__((__nothrow__)) __attribute__((__pure__)) __attribute__((__nonnull__(1
- static force_inline uint32_t DisassembleAt (struct DIS_fixed ∗result, uint32_t offset, uint32_t len)

7.3.1.1. Detailed Description

7.3.1.2. Define Documentation

7.3.1.2.1. #define DECLARE_SIGNATURE(name) Value:

```
const char *name##_sig;\
    __Signature name;
```

Declares a name for a subsignature

7.3.1.2.2. #define DEFINE_SIGNATURE(name, hex) Value:

```
.name##_sig = (hex),\
    .name = {__COUNTER__ - __signature_bias},
```

Defines the pattern for a previously declared subsignature.

See also:

DECLARE_SIGNATURE

Parameters:

name the name of a previously declared subsignature

hex the pattern for this subsignature

7.3.1.2.3. #define SIGNATURES_DECL_BEGIN struct __Signatures {

Marks the beginning of the subsignature name declaration section

7.3.1.2.4. #define SIGNATURES_DECL_END };

Marks the end of the subsignature name declaration section

7.3.1.2.5. #define SIGNATURES_DEF_BEGIN Value:

```
static const unsigned __signature_bias = __COUNTER__+1;\
const struct __Signatures Signatures = {\
```

Marks the beginning of subsignature pattern definitions.

See also:

SIGNATURES_DECL_BEGIN

7.3.1.2.6. #define SIGNATURES_END };\

Marks the end of the subsignature pattern definitions.

7.3.1.2.7. #define TARGET(tgt) const unsigned short __Target = (tgt);

Defines the ClamAV file target.

Parameters:

tgt ClamAV signature type (0 - raw, 1 - PE, etc.)

7.3.1.2.8. #define VIRUSNAME_PREFIX(name) const char __clambc_virusname_prefix[ ] = name;

Declares the virusname prefix.

Parameters:

name the prefix common to all viruses reported by this bytecode

7.3.1.2.9. #define VIRUSNAMES(...) const char *const __clambc_virusnames[ ] = {__VA_ARGS__};

Declares all the virusnames that this bytecode can report.

Parameters:

... a comma-separated list of strings interpreted as virusnames

7.3.1.3. Function Documentation

7.3.1.3.1. bool __is_bigendian (void) const

Returns true if the bytecode is executing on a big-endian CPU.

Returns:

true if executing on bigendian CPU, false otherwise

This will be optimized away in libclamav, but it must be used when dealing with endianess for portability reasons. For example whenever you read a 32-bit integer from a file, it can be written in little-endian convention (x86 CPU for example), or big-endian convention (PowerPC CPU for example). If the file always contains little-endian integers, then conversion might be needed. ClamAV bytecodes by their nature must only handle known-endian integers, if endianness can change, then both situations must be taken into account (based on a 1-byte field for example).

7.3.1.3.2. static uint16_t force_inline cli_readint16 (const void * *buff*) `[static]` Reads from the specified buffer a 16-bit of little-endian integer.

Parameters:

← ***buff*** pointer to buffer

Returns:

16-bit little-endian integer converted to host endianness

7.3.1.3.3. static uint32_t force_inline cli_readint32 (const void * *buff*) `[static]` Reads from the specified buffer a 32-bit of little-endian integer.

Parameters:

← ***buff*** pointer to buffer

Returns:

32-bit little-endian integer converted to host endianness

7.3.1.3.4. static void force_inline cli_writeint32 (void * *offset*, uint32_t *v*) `[static]` Writes the specified value into the specified buffer in little-endian order

Parameters:

→ ***offset*** pointer to buffer to write to

← ***v*** value to write

7.3.1.3.5. static force_inline uint32_t count_match (__Signature *sig*) [static] Returns how many times the specified signature matched.

Parameters:

sig name of subsignature queried

Returns:

number of times this subsignature matched in the entire file

This is a constant-time operation, the counts for all subsignatures are already computed.

7.3.1.3.6. static force_inline uint32_t DisassembleAt (struct DIS_fixed * *result*, uint32_t *offset*, uint32_t *len*) [static] Disassembles one X86 instruction starting at the specified offset.

Parameters:

→ ***result*** disassembly result

← ***offset*** start disassembling from this offset, in the current file

← ***len*** max amount of bytes to disassemble

Returns:

offset where disassembly ended

7.3.1.3.7. static force_inline void foundVirus (const char * *virusname*) [static] Sets the specified virusname as the virus detected by this bytecode.

Parameters:

virusname the name of the virus, excluding the prefix, must be one of the virusnames declared in `VIRUSNAMES`.

See also:

VIRUSNAMES

7.3.1.3.8. static force_inline uint32_t getEntryPoint (void) `[static]` Returns the offset of the EntryPoint in the executable file.

Returns:

offset of EP as 32-bit unsigned integer

7.3.1.3.9. static force_inline uint32_t getExeOffset (void) `[static]` Returns the offset of the executable in the file.

Returns:

offset of embedded executable inside file.

7.3.1.3.10. static force_inline uint32_t getFilesize (void) `[static]` Returns the currently scanned file's size.

Returns:

file size as 32-bit unsigned integer

7.3.1.3.11. static force_inline uint32_t getImageBase (void) `[static]` Returns the ImageBase with the correct endian conversion

7.3.1.3.12. static force_inline uint16_t getNumberOfSections (void) `[static]` Returns the number of sections in this executable file.

Returns:

number of sections as 16-bit unsigned integer

7.3.1.3.13. static force_inline uint32_t getPEBaseOfCode (void) `[static]` Return the PE BaseOfCode.

7.3.1.3.14. static force_inline uint32_t getPEBaseOfData (void) `[static]` Return the PE BaseOfData.

7.3.1.3.15. static force_inline uint16_t getPECharacteristics () [static]
Returns PE characteristics.

7.3.1.3.16. static force_inline uint32_t getPECheckSum (void) [static]
Return the PE CheckSum.

7.3.1.3.17. static force_inline uint32_t getPEDataDirRVA (unsigned *n*) [static] Gets the virtual address of specified image data directory.

Parameters:

n image directory requested

Returns:

Virtual Address of requested image directory

7.3.1.3.18. static force_inline uint32_t getPEDataDirSize (unsigned *n*) [static] Gets the size of the specified image data directory.

Parameters:

n image directory requested

Returns:

Size of requested image directory

7.3.1.3.19. static force_inline uint16_t getPEDllCharacteristics (void) [static] Return the PE DllCharacteristics.

7.3.1.3.20. static force_inline uint32_t getPEFileAlignment (void) [static] Return the PE FileAlignment.

7.3.1.3.21. static force_inline uint64_t getPEImageBase (void) [static]
Return the PE ImageBase as 64-bit integer.

7.3.1.3.22. static force_inline bool getPEisDLL () `[static]` Returns whether this is a DLL

7.3.1.3.23. static uint32_t getPELFANew (void) `[static]` Gets the offset to the PE header.

7.3.1.3.24. static force_inline uint32_t getPELoaderFlags (void) `[static]` Return the PE LoaderFlags.

7.3.1.3.25. static force_inline uint16_t getPEMachine () `[static]` Returns the CPU this executable runs on, see libclamav/pe.c for possible values

7.3.1.3.26. static force_inline uint16_t getPEMajorImageVersion (void) `[static]` Return the PE MajorImageVersion

7.3.1.3.27. static static force_inline force_inline uint8_t getPEMajorLinkerVersion (void) `[static]` Returns MajorLinkerVersion for this PE file.

7.3.1.3.28. static force_inline uint16_t getPEMajorOperatingSystemVersion (void) `[static]` Return the PE MajorOperatingSystemVersion.

7.3.1.3.29. static force_inline uint16_t getPEMajorSubsystemVersion (void) `[static]` Return the PE MajorSubsystemVersion

7.3.1.3.30. static force_inline uint16_t getPEMinorImageVersion (void) `[static]` Return the PE MinorImageVersion

7.3.1.3.31. static force_inline uint8_t getPEMinorLinkerVersion (void) `[static]` Returns MinorLinkerVersion for this PE file.

7.3.1.3.32. static force_inline uint16_t getPEMinorOperatingSystemVersion (void) `[static]` Return the PE MinorOperatingSystemVersion.

7.3.1.3.33. static force_inline uint16_t getPEMinorSubsystemVersion (void) [static] Return the PE MinorSubsystemVersion

7.3.1.3.34. static force_inline uint32_t getPENumberOfSymbols () [static] Returns the PE number of debug symbols

7.3.1.3.35. static force_inline uint32_t getPEPointerToSymbolTable () [static] Returns pointer to the PE debug symbol table

7.3.1.3.36. static force_inline uint32_t getPESectionAlignment (void) [static] Return the PE SectionAlignment.

7.3.1.3.37. static force_inline uint32_t getPESizeOfCode (void) [static] Return the PE SizeOfCode.

7.3.1.3.38. static force_inline uint32_t getPESizeOfHeaders (void) [static] Return the PE SizeOfHeaders.

7.3.1.3.39. static force_inline uint32_t getPESizeOfHeapCommit (void) [static] Return the PE SizeOfHeapCommit.

7.3.1.3.40. static force_inline uint32_t getPESizeOfHeapReserve (void) [static] Return the PE SizeOfHeapReserve.

7.3.1.3.41. static force_inline uint32_t getPESizeOfImage (void) [static] Return the PE SizeOfImage.

7.3.1.3.42. static force_inline uint32_t getPESizeOfInitializedData (void) [static] Return the PE SizeofInitializedData.

7.3.1.3.43. static force_inline uint16_t getPESizeOfOptionalHeader () [static] Returns the size of PE optional header.

7.3.1.3.44. static force_inline uint32_t getPESizeOfStackCommit (void) [static] Return the PE SizeOfStackCommit.

7.3.1.3.45. static force_inline uint32_t getPESizeOfStackReserve (void) [static] Return the PE SizeOfStackReserve.

7.3.1.3.46. static force_inline uint32_t getPESizeOfUninitializedData (void) [static] Return the PE SizeofUninitializedData.

7.3.1.3.47. static force_inline uint16_t getPESubsystem (void) [static] Return the PE Subsystem.

7.3.1.3.48. static force_inline uint32_t getPETimeDateStamp () [static] Returns the PE TimeDateStamp from headers

7.3.1.3.49. static force_inline uint32_t getPEWin32VersionValue (void) [static] Return the PE Win32VersionValue.

7.3.1.3.50. static force_inline bool hasExeInfo (void) [static] Returns whether the current file has executable information.

Returns:

true if the file has exe info, false otherwise

7.3.1.3.51. static force_inline bool isPE64 (void) [static] Returns whether this is a PE32+ executable.

Returns:

true if this is a PE32+ executable

7.3.1.3.52. static uint16_t force_inline le16_to_host (uint16_t *v*) [static] Converts the specified value if needed, knowing it is in little endian order.

Parameters:

$\leftarrow$ ***v*** 16-bit integer as read from a file

Returns:

integer converted to host's endianess

7.3.1.3.53. static uint32_t force_inline le32_to_host (uint32_t *v*) [static]
Converts the specified value if needed, knowing it is in little endian order.

Parameters:

← ***v*** 32-bit integer as read from a file

Returns:

integer converted to host's endianess

7.3.1.3.54. static uint32_t force_inline le64_to_host (uint32_t *v*) [static]
Converts the specified value if needed, knowing it is in little endian order.

Parameters:

← ***v*** 64-bit integer as read from a file

Returns:

integer converted to host's endianess

7.3.1.3.55. static force_inline uint32_t matches (__Signature *sig*) [static]
Returns whether the specified subsignature has matched at least once.

Parameters:

sig name of subsignature queried

Returns:

1 if subsignature one or more times, 0 otherwise

7.3.1.3.56. static void* memchr (const void * *s*, int *c*, size_t *n*) [static]
Scan the first `n` bytes of the buffer `s`, for the character `c`.

Parameters:

← ***s*** buffer to scan

c character to look for

n size of buffer

Returns:

a pointer to the first byte to match, or NULL if not found.

7.3.1.3.57. void∗ void∗ int memcmp (const void ∗ *s1*, const void ∗ *s2*, uint32_t *n*) Compares two memory buffers.

Parameters:

← ***s1*** buffer one

← ***s2*** buffer two

← ***n*** amount of bytes to copy

Returns:

an integer less than, equal to, or greater than zero if the first `n` bytes of `s1` are found, respectively, to be less than, to match, or be greater than the first `n` bytes of `s2`.

7.3.1.3.58. void∗ void∗ memcpy (void ∗restrict *dst*, const void ∗restrict *src*, uintptr_t *n*) Copies data between two non-overlapping buffers.

Parameters:

→ ***dst*** destination buffer

← ***src*** source buffer

← ***n*** amount of bytes to copy

Returns:

dst

7.3.1.3.59. void∗ memmove (void ∗ *dst*, const void ∗ *src*, uintptr_t *n*) Copies data between two possibly overlapping buffers.

Parameters:

→ ***dst*** destination buffer

← ***src*** source buffer

← ***n*** amount of bytes to copy

Returns:

dst

7.3.1.3.60. void∗ memset (void ∗ *src*, int *c*, uintptr_t *n*) Fills the specified buffer to the specified value.

Parameters:

→ ***src*** pointer to buffer

← ***c*** character to fill buffer with

← ***n*** length of buffer

Returns:

`src`

7.3.1.3.61. static force_inline int readPESectionName (unsigned char *name*[8], unsigned *n*) `[static]` Read name of requested PE section.

Parameters:

→ ***name*** name of PE section

← ***n*** PE section requested

Returns:

0 if successful, <0 otherwise

7.3.1.3.62. static force_inline bool readRVA (uint32_t *rva*, void ∗ *buf*, size_t *bufsize*) `[static]` read the specified amount of bytes from the PE file, starting at the address specified by RVA. Returns true on success (full read), false on any failure

CHAPTER 8

Copyright and License

8.1. 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.
```

```
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.
```

8.2. 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.

APPENDIX A
Predefined macros

```
#define __llvm__ 1
#define __clang__ 1
#define __GNUC_MINOR__ 2
#define __GNUC_PATCHLEVEL__ 1
#define __GNUC__ 4
#define __GXX_ABI_VERSION 1002
#define __VERSION__ "4.2.1 Compatible Clang Compiler"
#define __STDC__ 1
#define __STDC_VERSION__ 199901L
#define __STDC_HOSTED__ 0
#define __CONSTANT_CFSTRINGS__ 1
#define __CHAR_BIT__ 8
#define __SCHAR_MAX__ 127
#define __SHRT_MAX__ 32767
#define __INT_MAX__ 2147483647
#define __LONG_MAX__ 9223372036854775807L
#define __LONG_LONG_MAX__ 9223372036854775807LL
#define __WCHAR_MAX__ 2147483647
#define __INTMAX_MAX__ 9223372036854775807L
#define __INTMAX_TYPE__ long int
#define __UINTMAX_TYPE__ long unsigned int
#define __INTMAX_WIDTH__ 64
#define __PTRDIFF_TYPE__ int
#define __PTRDIFF_WIDTH__ 32
#define __INTPTR_TYPE__ long int
#define __INTPTR_WIDTH__ 64
#define __SIZE_TYPE__ unsigned int
#define __SIZE_WIDTH__ 32
#define __WCHAR_TYPE__ int
#define __WCHAR_WIDTH__ 32
#define __WINT_TYPE__ int
#define __WINT_WIDTH__ 32
#define __SIG_ATOMIC_WIDTH__ 32
#define __FLT_DENORM_MIN__ 1.40129846e-45F
#define __FLT_HAS_DENORM__ 1
#define __FLT_DIG__ 6
#define __FLT_EPSILON__ 1.19209290e-7F
#define __FLT_HAS_INFINITY__ 1
#define __FLT_HAS_QUIET_NAN__ 1
#define __FLT_MANT_DIG__ 24
#define __FLT_MAX_10_EXP__ 38
#define __FLT_MAX_EXP__ 128
#define __FLT_MAX__ 3.40282347e+38F
#define __FLT_MIN_10_EXP__ (-37)
#define __FLT_MIN_EXP__ (-125)
#define __FLT_MIN__ 1.17549435e-38F
#define __DBL_DENORM_MIN__ 4.9406564584124654e-324
#define __DBL_HAS_DENORM__ 1
#define __DBL_DIG__ 15
#define __DBL_EPSILON__ 2.2204460492503131e-16
#define __DBL_HAS_INFINITY__ 1
#define __DBL_HAS_QUIET_NAN__ 1
#define __DBL_MANT_DIG__ 53
#define __DBL_MAX_10_EXP__ 308
#define __DBL_MAX_EXP__ 1024
#define __DBL_MAX__ 1.7976931348623157e+308
#define __DBL_MIN_10_EXP__ (-307)
#define __DBL_MIN_EXP__ (-1021)
#define __DBL_MIN__ 2.2250738585072014e-308
#define __LDBL_DENORM_MIN__ 4.9406564584124654e-324
#define __LDBL_HAS_DENORM__ 1
```

```
#define __LDBL_DIG__ 15
#define __LDBL_EPSILON__ 2.2204460492503131e-16
#define __LDBL_HAS_INFINITY__ 1
#define __LDBL_HAS_QUIET_NAN__ 1
#define __LDBL_MANT_DIG__ 53
#define __LDBL_MAX_10_EXP__ 308
#define __LDBL_MAX_EXP__ 1024
#define __LDBL_MAX__ 1.7976931348623157e+308
#define __LDBL_MIN_10_EXP__ (-307)
#define __LDBL_MIN_EXP__ (-1021)
#define __LDBL_MIN__ 2.2250738585072014e-308
#define __POINTER_WIDTH__ 64
#define __INT8_TYPE__ char
#define __INT16_TYPE__ short
#define __INT32_TYPE__ int
#define __INT64_TYPE__ long int
#define __INT64_C_SUFFIX__ L
#define __USER_LABEL_PREFIX__ _
#define __FINITE_MATH_ONLY__ 0
#define __GNUC_STDC_INLINE__ 1
#define __NO_INLINE__ 1
#define __FLT_EVAL_METHOD__ 0
#define __FLT_RADIX__ 2
#define __DECIMAL_DIG__ 17
#define __CLAMBC__ 1
#define BYTECODE_API_H
#define __EXECS_H
#define BC_FEATURES_H
#define EBOUNDS(x)
#define __PE_H
#define DISASM_BC_H
#define __STDBOOL_H
#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
#define force_inline inline __attribute__((always_inline))
#define VIRUSNAME_PREFIX(name) const char __clambc_virusname_prefix[] = name;
#define VIRUSNAMES(...) const char *const __clambc_virusnames[] = {__VA_ARGS__};
#define PE_UNPACKER_DECLARE const uint16_t __clambc_kind = BC_PE_UNPACKER;
#define SIGNATURES_DECL_BEGIN struct __Signatures {
#define DECLARE_SIGNATURE(name) const char *name##_sig; __Signature name;
#define SIGNATURES_DECL_END };
#define TARGET(tgt) const unsigned short __Target = (tgt);
#define SIGNATURES_DEF_BEGIN static const unsigned __signature_bias = __COUNTER__+1;const struct
    __Signatures Signatures = {
#define DEFINE_SIGNATURE(name,hex) .name##_sig = (hex), .name = {__COUNTER__ - __signature_bias},
#define SIGNATURES_END };
#define RE2C_BSIZE 128
#define YYCTYPE unsigned char
#define YYCURSOR re2c_scur
#define YYLIMIT re2c_slim
#define YYMARKER re2c_smrk
#define YYCONTEXT re2c_sctx
#define YYFILL(n) { RE2C_FILLBUFFER(n); if (re2c_sres <= 0) break;}
#define REGEX_SCANNER unsigned char *re2c_scur, *re2c_stok, *re2c_smrk, *re2c_sctx, *re2c_slim; int
    re2c_sres; int32_t re2c_stokstart; unsigned char re2c_sbuffer[RE2C_BSIZE]; re2c_scur = re2c_slim =
    re2c_smrk = re2c_sctx = &re2c_sbuffer[0]; re2c_sres = 0; RE2C_FILLBUFFER(0);
#define REGEX_POS (-(re2c_slim - re2c_scur) + seek(0, SEEK_CUR))
#define REGEX_LOOP_BEGIN do { re2c_stok = re2c_scur; re2c_stokstart = REGEX_POS;} while (0);
#define REGEX_RESULT (re2c_sres)
#define RE2C_DEBUG_PRINT do { char buf[81]; uint32_t here = seek(0, SEEK_CUR); uint32_t d = re2c_slim -
    re2c_scur; uint32_t end = here - d; unsigned len = end - re2c_stokstart; if (len > 80) { unsigned
    skipped = len - 74; seek(re2c_stokstart, SEEK_SET); if (read(buf, 37) == 37) break; memcpy(buf+37,
    "[...]", 5); seek(end-37, SEEK_SET); if (read(buf, 37) != 37) break; buf[80] = '\0'; } else {
    seek(re2c_stokstart, SEEK_SET); if (read(buf, len) != len) break; buf[len] = '\0'; } buf[80] = '\0';
    debug_print_str(buf, 0); seek(here, SEEK_SET);} while (0)
#define DEBUG_PRINT_REGEX_MATCH RE2C_DEBUG_PRINT
#define BUFFER_FILL(buf,cursor,need,limit) do { (limit) = fill_buffer((buf), sizeof((buf)), (limit),
    (cursor), (need));} while (0);
#define BUFFER_ENSURE(buf,cursor,need,limit) do { if ((cursor) + (need) >= (limit)) { BUFFER_FILL(buf,
    cursor, need, limit) (cursor) = 0; }} while (0);
#define RE2C_FILLBUFFER(need) do { uint32_t cursor = re2c_stok - &re2c_sbuffer[0]; int32_t limit =
    re2c_slim - &re2c_sbuffer[0]; limit = fill_buffer(re2c_sbuffer, sizeof(re2c_sbuffer), limit,
    (cursor), (need)); if (!limit) { re2c_sres = 0; } else if (limit <= (need)) { re2c_sres = -1; } else
    { uint32_t curoff = re2c_scur - re2c_stok; uint32_t mrkoff = re2c_smrk - re2c_stok; uint32_t ctxoff =
    re2c_sctx - re2c_stok; re2c_slim = &re2c_sbuffer[0] + limit; re2c_stok = &re2c_sbuffer[0]; re2c_scur
    = &re2c_sbuffer[0] + curoff; re2c_smrk = &re2c_sbuffer[0] + mrkoff; re2c_sctx = &re2c_sbuffer[0] +
    ctxoff; re2c_sres = limit; }} while (0);
```