ClamAV Bytecode Compiler - Internals Manual

Török Edvin

March 28, 2010

Contents

ClamAV Bytecode Compiler - Internals Manual,

Authors: Török Edvin

CHAPTER 1

Overview

This manual describes internals details about the bytecode API, compiler, and libclamav bytecode interpreter/JIT. This manual is only of interest to ClamAV developers, see the ”ClamAV Bytecode Compiler User Manual” on how to write bytecode signatures.

CHAPTER 2

Bytecode libclamav hooks

2.1. Logical Signature hooks

2.2. PE hooks

2.3. Adding a new hook

A bytecode hook consists of the following:

- special global variables mapped to clamav internal structures,
- bytecode invoked at certain points in libclamav
- bytecode APIcalls specific to the hook

2.3.1. Adding new special globals for hooks

In the bytecode there are several special global variables named `__clambc_*`, which are mapped to libclamav internal variables.

These are globals from the bytecode's point of view to make bytecode writing easier, but they are not real globals in libclamav (it wouldn't be threadsafe). Instead in libclamav these "special globals" are stored in `struct cli_bc_ctx.hooks`, and the JIT/interpreter inserts special code to access fields of this struct as if they were globals.

Steps to add a new global to the bytecode compiler:

- Choose a unique name for the global (have a look at `clang/lib/Headers/bytecode_api.h`)

- Add a new value to `enum bc_global` in `ClamBC/clambc.h` named `GLOBAL_` followed by the uppercase name of the global. Make sure you add a new global before `_LAST_GLOBAL`, and don't change the order of the other enum values (this ensures that bytecodes that don't use the new global continue to work properly on old versions of libclamav that don't have the new global).

- Declare the global's name in `ClamBC/ClamBCModule.cpp`:

  `globalsMap["__clambc_<name>"] = GLOBAL_<NAME>;` where `<name>` and `<NAME>` are the lowercase/uppercase names of the global.

- Declare the new global in `clang/lib/Headers/bytecode_api.h`, order of declaration of globals doesn't matter here. The global must be declared as `extern const` and named __clambc_ followed by the lowercase name of the global.

- Run `./sync_clamav.sh` to generate `bytecode_api_decl.c.h`, `bytecode_api_impl.h`, `bytecode_hooks.h`.

Steps to add a new global to libclamav (needed if you add to compiler):

- In `libclamav/bytecode.c:cli_bytecode_context_alloc()` initialize the field of `ctx->hooks` corressponding to the new global

- Set the field coresponding to the global in the struct `ctx->hooks` in one of the API hooks, or introduce a new API hook that sets it.

- Note that the pointer set must be valid during the entire execution of the bytecode.

2.3.2. Adding new bytecode APIs

Bytecode APIs are external function calls from the bytecode into special entrypoints in libclamav.

To add a new API follow these steps:

- Add the prototype for the new API to `clang/lib/Headers/bytecode_api.h`, inside `#ifdef __CLAMBC__`

- Run `./sync_clamav.sh` to synchronize with libclamav

- Implement the new `cli_bcapi_` in `libclamav/bytecode_api.c`

- You can store values in fields of `ctx`, which is a hidden parameter, not accessible from bytecode.

- You can introduce new fields in `ctx` if needed to implement the API
- Do validation on input parameters, and any necessary security checks in the implementation of the API
- Create a new test in `examples/in/`, with the extension `.o1.c`, and update `sync_clamav.sh` to copy it to `unit_tests/input`
- Add a new testcase to `unit_tests/check_bytecode.c`:
    - Add a new `test_` function, and add it to the testcase with `tcase_add_test`
    - Call `cl_init` and `runtest` similar to other existing unit tests, but change the filename to the newly added unittest's name
    - Run make check, make sure it passes

CHAPTER 3

Updating LLVM

3.1. Update LLVM from upstream SVN

- `cd` into the git-svn dir of upstream LLVM
- Update LLVM [1]:

```
$ cd llvm
$ git svn fetch
$ git svn rebase --local
```

- Update clang:

```
$ cd clang
$ git svn fetch
$ git svn rebase --local
```

- Build it:

```
$ cd ../obj && ../llvm/configure --enable-optimized
$ make -j8
```

- All tests must pass before merging to clamav: `make check-all`
- (Optional) Build ClamAV with clang/x86 backend to test that the C frontend works:

```
$ cd /path/to/clamavsrc
$ ./configure CC=/path/to/clambc-compiler/obj/Release/bin/clang
$ make -j4
$ make check -j4
```

---

[1] this may require updating the svn-authors file

3.2. Merging LLVM to ClamAV bytecode compiler

Use the `merge-new.sh` script in the bytecode compiler repository. If there are no conflicts then the script takes care of merging, and comitting and.

If there are conflicts, the script will stop, and output an error message about the failed merge.

Fix the conflicts by using `git mergetool`, then commit the result using `git commit`.

Note that if llvm merge failed, clang is not merged either, so you should resume the merge of clang (easiest is to just rerun the script).

Then run `make check-all` for the compiler too.

Note: the script is now doing normal merges (i.e. unsquashed), to visualize just ”our” history use git log –first-parent

3.3. Merging LLVM to ClamAV (libclamav)

Update llvm remote: `git remote update llvm-upstream`.

Use the script `libclamav/c++/merge.sh` as above, from root of ClamAV source directory, there will be delete/modify conflicts.

Next run the script `libclamav/c++/strip-llvm.sh`, from the `libclamav/c++` directory, and see if there are any unneeded dirs left in LLVM. If there are, update the strip script, and rerun it. Now resolve any merge conflicts, commit the merge, and tag it as instructed by merge.sh.

Regenerate configure with autoconf 2.65:

- `cd llvm/autoconf`
- `sed -i '/Your/d' AutoRegen.sh`
- `./AutoRegen.sh`
- `git checkout AutoRegen.sh`
- `cd ..; git add configure; git add include/llvm/Config/config.h.in`

After the merge is complete, update the build files (if needed):

- do a Debug build of upstream LLVM
- Run `libclamav/c++/GenList.pl /path/to/llvm-objdir >out`
- Copy the _SOURCES definitions from `out` to `libclamav/c++/Makefile.am`

- Run automake in `libclamav/c++`
- Update the autogenerated files
- Build ClamAV
- Update to latest LLVM API (if needed)
- Build ClamAV
- Update win32 proj files: `win32/update-win32.pl --regen`

To update the autogenerated files:

- Configure ClamAV in maintainer mode [1]:
  ```
  ./configure --enable-maintainer-mode
  ```
- Build it:
  ```
  make -j8
  ```
- If tblgen fails to build, review the list of files in `tblgen_SOURCES`
- Review what files changed files (probably .inc and .gen files):
  ```
  git status
  ```
- Commit the result:
  ```
  git commit -a -m "Update autogenerated files after LLVM import"
  ```
- Fully clean the build dir [2]:
  ```
  git clean -xfd
  ```
- Test a normal (non-maintainer build, can be objdir != srcdir):
  ```
  ./configure && make && make check
  ```

Run `make check` from top-level builddir, this will run the LLVM tests too, make sure all of them pass.

Build ClamAV with `--enable-all-jit-targets` to test that all supported JIT targets build.

---

[1] Note that this must be a srcdir == objdir build

[2] Be careful to run this inside the ClamAV source dir, and not some other git repository

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.

The ClamAV bytecode backend translates from LLVM IR to ClamAV bytecode. Theoretically it could translate any LLVM IR which meets these constraints:

- No external function calls, except those defined by the ClamAV API
- No inline assembly
- ...

Thus (theoretically) any language that doesn't need an external language runtime (or the runtime can be compiled to the above restricted set of LLVM IR), could be compiled to ClamAV bytecode.

There are currently no plans currently to support any other language than C (maybe C++ when clang will support it).

4.1. Predefines

The following macros are predefined:

```
#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__ int
#define __INTPTR_WIDTH__ 32
#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_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 __FLT_HAS_DENORM__ 1
#define __DBL_DENORM_MIN__ 4.9406564584124654e-324
#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 __DBL_HAS_DENORM__ 1
#define __LDBL_DENORM_MIN__ 4.9406564584124654e-324
#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 __LDBL_HAS_DENORM__ 1
#define __POINTER_WIDTH__ 32
#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;
```

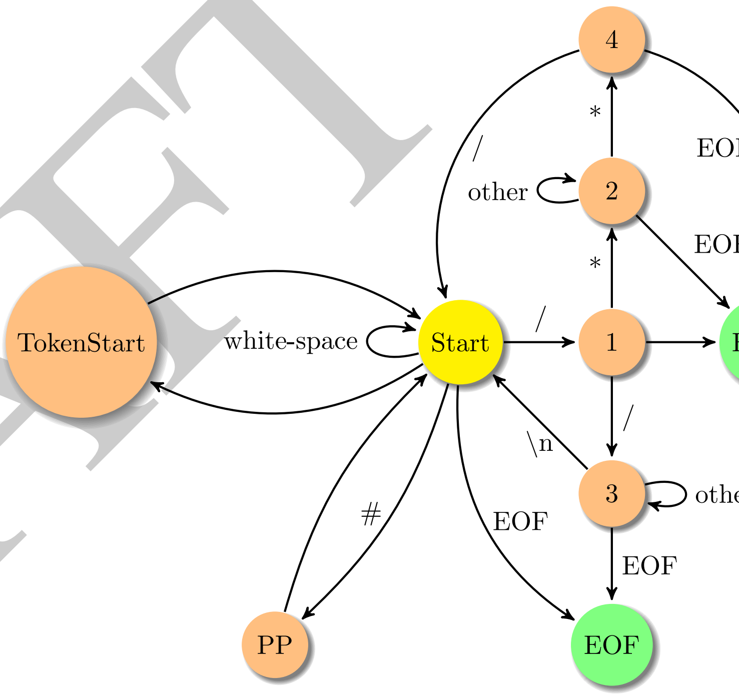
```
#define VIRUSNAMES(...) const char *const __clambc_virusnames[] = {__VA_ARGS__};
#define PE_UNPACKER_DECLARE const uint16_t __clambc_kind = BC_PE_UNPACKER;
#define SIGNATURES_DECL_BEGIN struct __Signatures {
#define DECLARE_SIGNATURE(name) const char *name##_sig; __Signature name;
#define SIGNATURES_DECL_END };
#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
#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; uns
#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; }} wh
#define RE2C_FILLBUFFER(need) do { uint32_t cursor = re2c_stok - &re2c_sbuffer[0]; int32_t limit = re2c_slim - &re2c_sbuffer[0]; limit = fill_bu
```

4.2. ClamAV API header restrictions

The ClamAV API header file (`bytecode_api.h`, and any files included by it) must be both valid C code, and conform to the following BNF grammar:

The reason is that the `ifacegen` program must be able to parse it to generate the api description, and glue code, and it only recognizes the above BNF grammar.

This also adds portability checks: any code conforming to that grammar should work properly both in the interpret and the JIT, even though a number of things have changed (such as sizeof int, which is why only fixed-size integers are allowed


4
/
*
EO
other
2
EO
*
TokenStart
white-space
Start
/
1
/
\n
3
othe
#
EOF
EOF
PP
EOF

CHAPTER 5

Publishing ClamAV bytecode

5.1. Pre-publish tests

The following tests are automatically performed prepublish:

- Compile the source code using the latest version of the ClamAV bytecode compiler (with user-specified optimization level):

```
$ clambc-compiler bytecode-726914.c -o testdir/bytecode-726914.cbc -O<N>
```

- Try to load the bytecode using the latest 2 stable version of ClamAV, both in JIT and interpreter mode [1]

```
$ export STABLEBIN=/usr/local/clamav-stable/bin
$ export DEVBIN=/usr/local/clamav-devel/bin
$ $STABLEBIN/clamscan -dtestdir/ -r /path/to/clamav-testfiles/
$ $DEVBIN/clamscan -dtestdir/ -r /path/to/clamav-testfiles/
$ $STABLEBIN/clamscan --force-interpreter -dtestdir/\
 -r /path/to/clamav-testfiles/
$ $DEVBIN/clamscan --force-interpreter -dtestdir/\
 -r /path/to/clamav-testfiles/
```

- Scan the sample(s) that will have this bytecode associated with the bytecode loaded (both interpreter and JIT mode):
- Scan the FPfarm

```
$ $STABLEBIN/clamscan -dtestdir/ -r /path/to/fpfarm/
$ $DEVBIN/clamscan -dtestdir/ -r /path/to/fpfarm/
```

---

[1] Since there is no stable version supporting bytecode, and the bytecode will be distributed in a separate cvd, for now we should test with latest nightly snapshot of ClamAV-devel. For 0.97 we should test with: 0.97, 0.96.1 (assuming those are latest 2 versions)

5.2. Building bytecode.cvd

Sigtool will perform some minimal checks on the bytecode prior to creating CVD:

- writes its own version in the header
- load the bytecode using libclamav API
- check that the interpreter and JIT can load it
- check that it is compilable to all configured targets (x86, ppc at least)
- check that the bytecode is production version (no debug metadata, all header fields are filled out, has associated virusname)

`%TODO: sigtool commandline`

CHAPTER 6

Copyright

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

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

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

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

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

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

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

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

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

Developed by:

    LLVM Team
```

University of Illinois at Urbana-Champaign

http://llvm.org

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

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

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

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

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

==

Copyrights and Licenses for Third Party Software Distributed with LLVM:

==

The LLVM software contains code written by third parties. Such software will have its own individual LICENSE.TXT file in the directory in which it appears. This file will describe the copyrights, license, and restrictions which apply to that code.

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

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

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