

ClamAV Bytecode Compiler - Internals Manual

Török Edvin

March 12, 2010

Contents

1	Overview	1
2	Bytecode libclamav hooks	3
2.1	Logical Signature hooks	3
2.2	PE hooks	3
2.3	Adding a new hook	3
2.3.1	Adding new special globals for hooks	3
2.3.2	Adding new bytecode APIs	4
3	Updating LLVM	7
3.1	Update LLVM from upstream SVN	7
3.2	Merging LLVM to ClamAV bytecode compiler	8
3.3	Merging LLVM to ClamAV (libclamav)	8
4	ClamAV bytecode language	11
4.1	Predefines	11
4.2	ClamAV API header restrictions	13
5	Publishing ClamAV bytecode	15
5.1	Pre-publish tests	15
5.2	Building bytecode.cvd	16
6	Copyright	17

ClamAV Bytecode Compiler - Internals Manual,

© 2009 Sourcefire, Inc.

Authors: Török Edvin

This document is distributed under the terms of the GNU General Public License v2.

Clam AntiVirus is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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.

ClamAV and Clam AntiVirus are trademarks of Sourcefire, Inc.

CHAPTER 1

Overview

This manual describes internal 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.

DRAFT

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 API calls 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` corresponding to the new global
- Set the field corresponding 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 entry-points 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

DRAFT

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/clamav-compiler/obj/Release/bin/clang
$ make -j4
$ make check -j4
```

¹this 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, committing and tagging.

If there are conflicts, the script will stop, and output a message:

Merge failed: resolve conflicts and run: `git tag merge-llvm-<REV>`

Note down the `git tag` command, fix the conflicts by using `git mergetool`, then commit the result using `git commit`. The squash commit message is in `.git/SQUASH_MSG` in this case, you should paste that into the commit message. Finally run the `git tag` command you noted down.

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.

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.

¹Note that this must be a `srcdir == objdir` build

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

DRAFT

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:

```
1 #define __llvm__ 1
   #define __clang__ 1
3 #define __GNUC_MINOR__ 2
   #define __GNUC_PATCHLEVEL__ 1
5 #define __GNUC__ 4
   #define __GXX_ABI_VERSION 1002
7 #define __VERSION__ "4.2.1 Compatible Clang Compiler"
   #define __STDC__ 1
9 #define __STDC_VERSION__ 199901L
   #define __STDC_HOSTED__ 0
11 #define __CONSTANT_CFSTRINGS__ 1
   #define __CHAR_BIT__ 8
13 #define __SCHAR_MAX__ 127
   #define __SHRT_MAX__ 32767
15 #define __INT_MAX__ 2147483647
   #define __LONG_MAX__ 9223372036854775807L
```

```

17 #define _LONG_LONG_MAX_ 9223372036854775807LL
   #define _WCHAR_MAX_ 2147483647
19 #define _INTMAX_MAX_ 9223372036854775807L
   #define _INTMAX_TYPE_ long int
21 #define _UINTMAX_TYPE_ long unsigned int
   #define _INTMAX_WIDTH_ 64
23 #define _PTRDIFF_TYPE_ int
   #define _PTRDIFF_WIDTH_ 32
25 #define _INTPTR_TYPE_ int
   #define _INTPTR_WIDTH_ 32
27 #define _SIZE_TYPE_ unsigned int
   #define _SIZE_WIDTH_ 32
29 #define _WCHAR_TYPE_ int
   #define _WCHAR_WIDTH_ 32
31 #define _WINT_TYPE_ int
   #define _WINT_WIDTH_ 32
33 #define _SIG_ATOMIC_WIDTH_ 32
   #define _FLT_DENORM_MIN_ 1.40129846e-45F
35 #define _FLT_DIG_ 6
   #define _FLT_EPSILON_ 1.19209290e-7F
37 #define _FLT_HAS_INFINITY_ 1
   #define _FLT_HAS_QUIET_NAN_ 1
39 #define _FLT_MANT_DIG_ 24
   #define _FLT_MAX_10_EXP_ 38
41 #define _FLT_MAX_EXP_ 128
   #define _FLT_MAX_ 3.40282347e+38F
43 #define _FLT_MIN_10_EXP_ (-37)
   #define _FLT_MIN_EXP_ (-125)
45 #define _FLT_MIN_ 1.17549435e-38F
   #define _FLT_HAS_DENORM_ 1
47 #define _DBL_DENORM_MIN_ 4.9406564584124654e-324
   #define _DBL_DIG_ 15
49 #define _DBL_EPSILON_ 2.2204460492503131e-16
   #define _DBL_HAS_INFINITY_ 1
51 #define _DBL_HAS_QUIET_NAN_ 1
   #define _DBL_MANT_DIG_ 53
53 #define _DBL_MAX_10_EXP_ 308
   #define _DBL_MAX_EXP_ 1024
55 #define _DBL_MAX_ 1.7976931348623157e+308
   #define _DBL_MIN_10_EXP_ (-307)
57 #define _DBL_MIN_EXP_ (-1021)
   #define _DBL_MIN_ 2.2250738585072014e-308
59 #define _LDBL_DENORM_MIN_ 4.9406564584124654e-324
   #define _LDBL_DIG_ 15
61 #define _LDBL_EPSILON_ 2.2204460492503131e-16
   #define _LDBL_HAS_INFINITY_ 1
63 #define _LDBL_HAS_QUIET_NAN_ 1
   #define _LDBL_MANT_DIG_ 53
65 #define _LDBL_MAX_10_EXP_ 308
   #define _LDBL_MAX_EXP_ 1024
67 #define _LDBL_MAX_ 1.7976931348623157e+308
   #define _LDBL_MIN_10_EXP_ (-307)
69 #define _LDBL_MIN_EXP_ (-1021)
   #define _LDBL_MIN_ 2.2250738585072014e-308
71 #define _LDBL_HAS_DENORM_ 1
   #define _POINTER_WIDTH_ 32
73 #define _INT8_TYPE_ char
75 #define _INT16_TYPE_ short
   #define _INT32_TYPE_ int
77 #define _INT64_TYPE_ long int
   #define _INT64_C_SUFFIX_ L
79 #define _USER_LABEL_PREFIX_ _
   #define _FINITE_MATH_ONLY_ 0
81 #define _GNUC_STDC_INLINE_ 1
   #define _NO_INLINE_ 1
83 #define _FLT_EVAL_METHOD_ 0
   #define _FLT_RADIX_ 2
85 #define _DECIMAL_DIG_ 17
   #define _CLAMBC_ 1
87 #define BYTECODE_API_H
   #define _EXECS_H
89 #define BC_FEATURES_H
   #define EBOUNDS(fieldname) __attribute__((bounds(fieldname)))
91 #define _PE_H
   #define DISASM_BC_H
93 #define _STDBOOL_H
   #define bool _Bool
95 #define true 1
   #define false 0
97 #define __bool_true_false_are_defined 1
   #define force_inline inline __attribute__((always_inline))
99 #define VIRUSNAME_PREFIX(name) const char __clambc_virusname_prefix[] = name;

```

```

#define VIRUSNAMES(...) const char *const __clambc_virusnames[] = {__VA_ARGS__};
101 #define PE.UNPACKER.DECLARE const uint16_t __clambc_kind = BC.PE.UNPACKER;
#define SIGNATURES.DECL.BEGIN struct __Signatures {
103 #define DECLARE.SIGNATURE(name) const char *name##_sig; __Signature name;
#define SIGNATURES.DECL.END };
105 #define TARGET(tgt) const unsigned short __Target = (tgt);
#define SIGNATURES.DEF.BEGIN static const unsigned __signature_bias = __COUNTER__+1; const struct __Signatures Signatures = {
107 #define DEFINE.SIGNATURE(name,hex) .name##_sig = (hex), .name = {__COUNTER__ - __signature_bias},
#define SIGNATURES.END };
109 #define RE2C.BSIZE 128
#define YYCTYPE unsigned char
111 #define YYCURSOR re2c_scur
#define YYLIMIT re2c_slim
113 #define YYMARKER re2c_smrk
#define YYCTXMARKER re2c_sctx
115 #define YYFILL(n) { RE2C.FILLBUFFER(n) if (re2c_sres >= 0) break; }
#define REGEX.SCANNER unsigned char *re2c_scur, *re2c_slim, *re2c_smrk, *re2c_sctx, *re2c_seof, *re2c_stok; int re2c_sres; int32_t re2c_stoksta
117 #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);
119 #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
121 #define DEBUG.PRINT.REGEX.MATCH RE2C.DEBUG.PRINT
#define RE2C.FILLBUFFER(len) { if (!re2c_seof) { int got, cnt = re2c_stok - re2c_sbuffer; if (cnt > re2c_slim - re2c_sbuffer) { cnt = 0; re2c.s

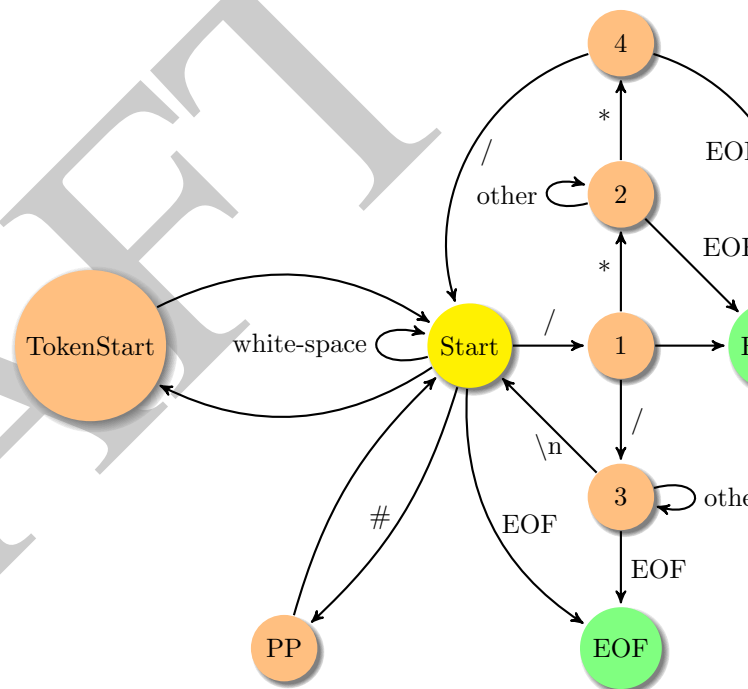
```

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



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/
```

¹Since 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}