llvm code generate and JIT execute in clamav

by eqmcc

eqmcc@http://blog.csdn.net/eqmcc

Description

in bytecode scan mode for clamav, the bytecode is IR code of llvm and will be written in a 'c-like' language and compiled to IR bytecode using clamav-bytecode-compiler. Later, in virus scan run time, IR form bytecode will be firstly restored into cli_bc ta structure in clamav, then passed to llvm , later rebuilt to in-memory IR code and finally compiled to local machine code with exection.

this document will talk about how the c like bytecode is compiled to IR code and later executed In JIT.

Data structures

Test case

1. source-code and bytecode

1.1 source code

test_bytecode.c

```
VIRUSNAME_PREFIX("test_bytecode")
VIRUSNAMES("A","B")
TARGET(7)
SIGNATURES_DECL_BEGIN
DECLARE_SIGNATURE(magic)
SIGNATURES_DECL_END

SIGNATURES_DEF_BEGIN
DEFINE_SIGNATURE(magic,"61616262")        // the pattern as "aabb" in hex
SIGNATURES_END
bool logical_trigger (void)
{
        // @ clamav-bytecode-compiler/obj/Release/lib/clang/1.1/include/bytecode_local.h
        return count_match(Signatures.magic) != 1; // if "aabb" match count is '1', it's not a virus
}

int entrypoint (void)
{
        int count = count_match(Signatures.magic);
        if ( count == 3)    foundVirus("B"); // 3 matches of "aabb", find virus B
        else foundVirus ("A"); // other case, find virus A
        return 0;
}
```

1.2 source code

1.2.1 compile

compile the source code to bytecode file *test_bytecode.cbc* via following command:

```
clambc-compiler test_bytecode.c -o test_bytecode.cbc -O2
```

and the *test_bytecode.cbc* looks as bellow:

```
ClamBCafhndcbn`ae|aeegcgefbg```c``a```|bjacflfafmfbfcfmb`cnbicgcnbccafmbecmbgfffecdfdfacdfcc``bcaaap`clamcoincidencejb:4096
test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
Teddaaahdabahdacahdadahdaeahdafahdagahebbeebaeebcdebadaacb`bbadb`bdb`aahdb`db`b
Eaeaaaeb`e|amcgefdgfgifbgegcgnfafmfef``
Gd```hai`@`b`aC``a`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBad@`baeBdgBefBcgBdgBoeBbfBigBdgBefBcfBofBdfBefBnbBbd@`bcd@D```h`bad@A
b`bad@Ab`bad@Ac`bad@Ac`
A`b`bLadb`b`aa`b`b`b`b`Fagac
Bb`b`gbAd`aaaaeab`b`AcdTaaaaaaab
Bb`bababbaeAh`AodTcab`b@d
Bb`bacabbaeAf`AodTcab`b@dE
Sfeidbeeecendadmdedoe`ebeedfdidhehbbbdgefcgdgoebfigdgefcfofdfefbbibSfeidbeeecendadmdedcehbbbadbblbbbbdbbibSdeadbegdeddehbgcibSceidgdnda
ddeeebeedceoeddedcdldoebdedgdidnd
ddedcdldadbeedoeceidgdndaddeeebeedhbmfafgfifcfibSceidgdndaddeeebeedceoeddedcdldoeedndddSSceidgdndaddeeebeedceoeddedfdoebdedgdidndSdde
dfdidndedoeceidgdndaddeeebeedhbmfafgfifcflbbbfcacfcacfcbcfcbcbbib`b`bobob`bdghfef`b`gafdgdgefbgnf`bafcg`bgbgbafafbfbfbb`bifnf`bhfefhg
ceidgdndaddeeebeedceoeedndddSbfofoflf`blfofgfifcfaflfoedgbgifgfgfefbg`bhbfgofifdfibSkgSobob`b`d`bcflfafmfaffgmbbfigdgefcfofdfefmbcfofmf`giflfefbgobofb
fjfobbeeflfefafcgefoblfifbfobcflfafnfgfobacnbacobifnfcflfegdfefobbfigdgefcfofdfefoelfofcfaflfnbhf
bgefdgegbgnf`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfib`babmc`backcSmgSSifnfdg`befnfdgbgig`gofifnfdg`bhbfgofifdfibSkgSifnfdg`bcfofeg
nfdg`bmc`bcfofegnfdgoemfafdgcfhfhbceifgfnfafdgegbgefcgnbmfafgfifcfibkc
ifff`bhb`bcfofegnfdg`bmcmc`bccib`b`bffofegnfdffeifbgegcghbbbbdbbibkc`bobob`bcc`bmfafdgcfhfefcg`bofff`bblciafafbfbfbldilb`bffifnfdf`bfgifbgegcg`bbdSeflfcg
ef`bffofegnfdffeifbgegcg`bhbbbadbbibkc`bobob`bofdghfefbg`bcfafcgeflb`bffifnfdf`bfgifbgegcg`bad
bgefdgegbgnf`b`ckcSmgSS
```

1.2.2 verify

verify the bytecode info via following command:

```
clambc --info test_bytecode.cbc
```

and output as bellow

```
Bytecode format functionality level: 6
Bytecode metadata:
        compiler version: clambc-0.97.3a-5-gf5dd1d3
        compiled on: (1359881038) Sun Feb   3 03:43:58 2013
        compiled by: user
        target exclude: 0
        bytecode type: logical only
        bytecode functionality level: 0 - 0
        bytecode logical signature: test_bytecode.{A,B};Engine:56-255,Target:7;((0<1)|(0>1));61616262
        // ((0<1)|(0>1))means the logic signature will be matched if thesub logic sig’s match count is not 1
        virusname prefix: (null)
        virusnames: 0
        bytecode triggered on: files matching logical signature
        number of functions: 1
```

```
number of types: 19
number of global constants: 9
number of debug nodes: 0
bytecode APIs used:
 setvirusname
```

2. run test

2.1 test files

test3.txt

```
aabbxxxxxxxxxxxxxxxxxxxxaabbxxxxxxxxxxxxaabb
```

2.2 test run

```
clamscan --bytecode=yes --bytecode-unsigned=yes  -d test_bytecode.cbc test3.txt
```

comments:
--bytecode=yes : enable bytecode scan
--bytecode-unsigned=yes : load unofficial bytecode

2.2.2 result

```
user@ubuntu:~/clamav$ clamscan --bytecode=yes --bytecode-unsigned=yes test2.txt
test3.txt: test_bytecode.B FOUND
----------- SCAN SUMMARY -----------
Known viruses: 1
Engine version: devel-a62bf02
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.00 MB
Data read: 0.00 MB (ratio 0.00:1)
Time: 0.074 sec (0 m 0 s)
```

compile - from c-like code to IR code(cbc file)

the test run to build:

```
mkdir obj
cd obj
../llvm/configure --enable-optimized --enable-targets=host-only --disable-bindings --prefix=/usr/local/clamav
make clambc-only -j4;sudo make install-clambc -j8
clambc-compiler test_bytecode.c -o test_bytecode.cbc -O2
```

data structure

frontend action type

```
enum ActionKind {
    ASTDump,                ///< Parse ASTs and dump them.
    ASTPrint,               ///< Parse ASTs and print them.
    ASTPrintXML,            ///< Parse ASTs and print them in XML.
    ASTView,                ///< Parse ASTs and view them in Graphviz.
    DumpRawTokens,          ///< Dump out raw tokens.
    DumpRecordLayouts,      ///< Dump record layout information.
    DumpTokens,             ///< Dump out preprocessed tokens.
    EmitAssembly,           ///< Emit a .s file.
    EmitBC,                 ///< Emit a .bc file.
    EmitHTML,               ///< Translate input source into HTML.
    EmitLLVM,               ///< Emit a .ll file.
    EmitLLVMOnly,           ///< Generate LLVM IR, but do not
    EmitObj,                ///< Emit a .o file.
    FixIt,                  ///< Parse and apply any fixits to the source.
    GeneratePCH,            ///< Generate pre-compiled header.
    GeneratePTH,            ///< Generate pre-tokenized header.
    InheritanceView,        ///< View C++ inheritance for a specified class.
    ParseNoop,              ///< Parse with noop callbacks.
    ParsePrintCallbacks,    ///< Parse and print each callback.
    ParseSyntaxOnly,        ///< Parse and perform semantic analysis.
    PluginAction,           ///< Run a plugin action, \see ActionName.
    PrintDeclContext,       ///< Print DeclContext and their Decls.
    PrintPreprocessedInput, ///< -E mode.
    RewriteMacros,          ///< Expand macros but not #includes.
    RewriteObjC,            ///< ObjC->C Rewriter.
    RewriteTest,            ///< Rewriter playground
    RunAnalysis,            ///< Run one or more source code analyses.
    RunPreprocessorOnly     ///< Just lex, no output.
};
```

call flow:

start from "clamav-bytecode-compiler/driver/main/main.cpp":

main
    CompileFile
        CompileSubprocess
            compileInternal

before compile, will merge all other api source etc.. and do a token identification: Preprocessor::LookUpIdentifierInfo.

the compile uses clang as the c-like bytecode's frontend, in CompileSubprocess:

```
/// CompilerInstance - Helper class for managing a single instance of the Clang
/// compiler.
///
/// The CompilerInstance serves two purposes:
///    (1) It manages the various objects which are necessary to run the compiler,
///          for example the preprocessor, the target information, and the AST
///          context.
///    (2) It provides utility routines for constructing and manipulating the
///          common Clang objects.
///
/// The compiler instance generally owns the instance of all the objects that it
/// manages. However, clients can still share objects by manually setting the
/// object and retaking ownership prior to destroying the CompilerInstance.
///
/// The compiler instance is intended to simplify clients, but not to lock them
/// in to the compiler instance for everything. When possible, utility functions
/// come in two forms; a short form that reuses the CompilerInstance objects,
/// and a long form that takes explicit instances of any required objects.
CompilerInstance Clang;
Clang.setLLVMContext(new llvm::LLVMContext);
```

eqmcc@

JIT - rebuild IR code and compile into machine code

bytecode IR(s) are passed into llvm from clamav via a function call int cli_bytecode_prepare_jit(struct cli_all_bc *bcs) with all loaded bytecode(s). The structure cli_all_bc contains an array of cli_bc data structure which holds all the info needed for a bytecode to be rebuilt(remember in clamav's bytecode loading function, the bytecode cbc file - IR code - is parsed and all the info of a bytecode is filled into the structure cli_bc).

in cli_bytecode_prepare_jit, information necessary for rebuilding the bytecode is abstracted and IR code is regenerated via various create methods of related data structure which stands for different component of a procedure code. Later, IR code will be complied and pointer to local machine code generated will be logged when JIT mode is enabled and execution engine is initialized.

Note:

for IR(intermediate representation) code, there are three forms, all equivalent to each other:

in-memory IR code

file IR code(.cb file) - clang -emit-llvm -c hello.c -o hello.bc

readable IR code(.ll file) - llvm-dis hello.bc

So basically in the whole process is:

1. c-like bytecode is compiled to llvm IR(.bc) code via clamav-bytecode-compiler
2. cli_loadcbc function, IR code is parsed into structure called cli_bc
3. cli_bc structure is passed into llvm via function call to cli_bytecode_prepare_jit and in this very function, in-memory IR code will be rebuilt according the information at cli_bc and functions will be compiled to local machine code on the fly if JIT mode is needed
4. run the local machine code designated by clamav function call cli_vm_execute_jit

in rest of this chapter, how IR code is rebuilt and how IR code is compiled to local machine code will be investigated.

rebuild (in-memory)IR code from cli_bc information

the job of rebuilding IR and compiling IR is done at cli_bytecode_prepare_jit

```
// define an execute engine for JIT compile
ExecutionEngine *EE = bcs->engine->EE = builder.create();

// for api functions, this is an array of piointers
Function **apiFuncs = new Function *[cli_apicall_maxapi];
/// addGlobalMapping - Tell the execution engine that the specified global is
/// at the specified location.    This is used internally as functions are JIT'd
/// and as global variables are laid out in memory. It can and should also be
/// used by clients of the EE that want to have an LLVM global overlay
/// existing data in memory.
// so meaning of bellow call is two folded:
//1. tell EE the location of a var/function
```

eqmcc@http://blog.csdn.net/eqmcc

```
//2. overlay global var/function's address
EE->addGlobalMapping(F, dest);
/// getPointerToFunction - This method is used to get the address of the
/// specified function, compiling it if neccesary.
EE->getPointerToFunction(F);
apiFuncs[i] = F;

// for bytecode functions, this is an array of pionters
llvm::Function **Functions = new Function*[bcs->count];
// rebuilt the IR code from cli_bc
LLVMCodegen Codegen(bc, M, &CF, bcs->engine->compiledFunctions, EE,
                    OurFPM, OurFPMUnsigned, apiFuncs, apiMap);
// rebuilt and compile
Function *F = Codegen.generate();
Functions[i] = F; // F is a wrapper which will call functions in a bytecode
```

actual rebuilt and compile of function's IR code is at LLVMCodegen::generate()

a function in llvm consists of:

return value(type and value)

arguments(type and value)

global vars(type and value)

local vars(type and value)

function(declaration and instructions)

Global

```
LLVMCodegen::globals; // store all globals in a function - std::vector<Value*>
GlobalVariable *GV; // define a global
Constant *C = buildConstant(Ty, bc->globals[i], c); // build a constant for global value
// create the global
GV = new GlobalVariable(*M, Ty, true,
                        GlobalValue::InternalLinkage,
                        C, "glob"+Twine(i));
// store the global in stack
globals.push_back(GV);
```

function

+agr type

```
std::vector<constType*> argTypes;
argTypes.push_back(HiddenCtx);
argTypes.push_back(mapType(func->types[a]));
```

+ret type

```
constType *RetTy = mapType(func->returnType);
```

+function type

```
FunctionType *FTy =    FunctionType::get(RetTy, argTypes, false);
```

+declare function

```
Functions[j] = Function::Create(FTy, Function::InternalLinkage, BytecodeID+"f"+Twine(j), M);
```

+Basic Block

```
BasicBlock **BB = new BasicBlock*[func->numBB];
BB[i] = BasicBlock::Create(Context, "", F); // create BB
```

+value

```
/// Value Class
/// This is a very important LLVM class. It is the base class of all values
/// computed by a program that may be used as operands to other values.
Values = new Value*[func->numValues];
//this table have values for all arguments and local variables
```

++ arguments' value

```
for (unsigned i=0;i<func->numArgs; i++)
      // arguments
      Values[i] = &*I;

//in parseFunctionHeader
//func->numValues = func->numArgs + func->numLocals;
```

++ local variables' value

```
for (unsigned i=func->numArgs;i<func->numValues;i++)
      // local variables
      Values[i] = Builder.CreateAlloca(mapType(func->types[i]));
```

Basic Block - example on how ADD(OP_BC_ADD) instruction is implemented

Store function will insert specific instruction into a BB via IRBuilderDefaultInserter::InsertHelper

eqmcc@http://blog.csdn.net/eqmcc

```
Store(inst->dest, Builder.CreateAdd(Op0, Op1));
    // Values is memory space for local vars and agrs
    Builder.CreateStore(V, Values[dest]);
        return Insert(new StoreInst(Val, Ptr, isVolatile));
            /// Insert - Insert and return the specified instruction.
            InstTy *Insert(InstTy *I, const Twine &Name = "") const
          /// IRBuilderDefaultInserter - This provides the default implementation of the
          /// IRBuilder 'InsertHelper' method that is called whenever an instruction is
          /// created by IRBuilder and needs to be inserted.    By default, this inserts the
          /// instruction at the insertion point.
              IRBuilderDefaultInserter::InsertHelper(I, Name, BB, InsertPt);
              if (BB) BB->getInstList().insert(InsertPt, I); // insert in current BB

Builder.CreateAdd(Op0, Op1)
    Insert(BinaryOperator::CreateAdd(LHS, RHS), Name);
        BinaryOperator::CreateAdd
```

this CreateAdd function is generated as bellow：

```
/// Create* - These methods just forward to Create, and are useful when you
/// statically know what type of instruction you're going to create.    These
/// helpers just save some typing.
#define HANDLE_BINARY_INST(N, OPC, CLASS) \
   static BinaryOperator *Create##OPC(Value *V1, Value *V2, \ const Twine &Name = "") {\
      return Create(Instruction::OPC, V1, V2, Name);\
   }
```

finally will call BinaryOperator::Create as bellow which will init a *BinaryOperator*:

```
+ create BinaryOperator
BinaryOperator *BinaryOperator::Create(BinaryOps Op, Value *S1, Value *S2,
                                       const Twine &Name,
                                       Instruction *InsertBefore) {
   assert(S1->getType() == S2->getType() &&
          "Cannot create binary operator with two operands of differing type!");
   return new BinaryOperator(Op, S1, S2, S1->getType(), Name, InsertBefore);
}
```

the initialization of a BinrayOperator looks as bellow:

```
BinaryOperator::BinaryOperator(BinaryOps iType, Value *S1, Value *S2,
                               const Type *Ty, const Twine &Name,
                               Instruction *InsertBefore)
   : Instruction(Ty, iType,
                 OperandTraits<BinaryOperator>::op_begin(this),
                 OperandTraits<BinaryOperator>::operands(this),
                 InsertBefore) {
```

```
    Op<0>() = S1;
    Op<1>() = S2;
    init(iType);
    setName(Name);
}
```

this BinaryOperator initialization will further initialize Instruction::Instruction as bellow:

```
+ create and insert Instruction
Instruction::Instruction(const Type *ty, unsigned it, Use *Ops, unsigned NumOps,
                         Instruction *InsertBefore)
  : User(ty, Value::InstructionVal + it, Ops, NumOps), Parent(0) {
  // Make sure that we get added to a basicblock
  LeakDetector::addGarbageObject(this);

  // If requested, insert this instruction into a basic block...
  if (InsertBefore) {
    assert(InsertBefore->getParent() &&
           "Instruction to insert before is not in a basic block!");
    InsertBefore->getParent()->getInstList().insert(InsertBefore, this);
  }
}
```

emit local machine code from in-memory IR code

compile IR code to local machine code can be achieved by either running function pass or calling getPointerToFunction(this one will actually ending at running function pass as well)

preparation of code emitting

before everything, certain jobs need to be done to prepare the necessary initialization of component of emitting the machine codes, i.e.: adding the passes necessary to emit the code, Quote from “Life of an instruction in LLVM”:

> *The sequence of passes to JIT-emit code is defined by LLVMTargetMachine::addPassesToEmitMachineCode. It calls addPassesToGenerateCode, which defines all the passes required to do what most of this article has been talking about until now – turning IR into MI form. Next, it calls addCodeEmitter, which is a target-specific pass for converting MIs into actual machine code. Since MIs are already very low-level, it’s fairly straightforward to translate them to runnable machine code [8]. The x86 code for that lives in lib/Target/X86/X86CodeEmitter.cpp. For our division instruction there’s no special handling here, because the MachineInstr it’s packaged in already contains its opcode and operands. It is handled generically with other*

eqmcc@http://blog

instructions in emitInstruction.

when creating ExecutionEngine, a JIT instance will be created which will call addPassesToEmitMachineCode which does the preparation:

```
ExecutionEngine *EngineBuilder::create()
   ExecutionEngine::JITCtor(M, ErrorStr, JMM, OptLevel,
                                        AllocateGVsWithCode, CMModel,
                                        MArch, MCPU, MAttrs);
```

libclamav/c++/llvm/lib/ExecutionEngine/JIT/JIT.h: JITCtor = createJIT;

```
   ExecutionEngine *ExecutionEngine::createJIT
      ExecutionEngine *JIT::createJIT
         // Pick a target either via -march or by guessing the native arch.
         TargetMachine *TM = JIT::selectTarget(M, MArch, MCPU, MAttrs, ErrorStr);
         new JIT(M, *TM, *TJ, JMM, OptLevel, GVsWithCode);
            JIT::JIT
               jitstate = new JITState(M);
               /// JITEmitter - The JIT implementation of the MachineCodeEmitter,
         which is used to output functions to memory for execution.
               JCE = createEmitter(*this, JMM, TM);
                // Add target data
                MutexGuard locked(lock);
                FunctionPassManager &PM = jitstate->getPM(locked);
                PM.add(new TargetData(*TM.getTargetData()));

                //Turn the machine code intermediate representation into bytes in
                //memory that may be executed.
               TM.addPassesToEmitMachineCode(PM, *JCE, OptLevel)
```

LLVMTargetMachine::addPassesToEmitMachineCode

```
/// addPassesToEmitMachineCode - Add passes to the specified pass manager to
/// get machine code emitted.   This uses a JITCodeEmitter object to handle
/// actually outputting the machine code and resolving things like the address
/// of functions.   This method should returns true if machine code emission is
/// not supported.
bool LLVMTargetMachine::addPassesToEmitMachineCode(PassManagerBase &PM,
                                                   JITCodeEmitter &JCE,
                                                   CodeGenOpt::Level OptLevel,
                                                   bool DisableVerify) {
   // Make sure the code model is set.
     setCodeModelForJIT();

   // Add common CodeGen passes.
     MCContext *Ctx = 0;
     if (addCommonCodeGenPasses(PM, OptLevel, DisableVerify, Ctx))
        return true;
```

```
    addCodeEmitter(PM, OptLevel, JCE);
    PM.add(createGCInfoDeleter());

    return false; // success!
}
```

LLVMTargetMachine::addCommonCodeGenPasses

```
/// addCommonCodeGenPasses - Add standard LLVM codegen passes used for both
/// emitting to assembly files or machine code output.
```

compile and emit code

first, in LLVMCodegen::generate(), after a function's IR code is fully generated, FunctionPassManager::run(i.e.: PM.run(*F);) will be called to run all the function passes that is registered which includes the pass to generate the local machine code of the functions.
also follow up on above, for a bytecode code defined for specific purpose, no matter how many functions are defined in it, all these functions will be called accordingly with a entry point(in bytecode, it's enrtypoint function). So in LLVMCodegen::generate, after all code function are rebuilt/compiled, the entry point function will be wrapped up and stands for the entry point to be registered in the llvm execution engine which may be called in later virus scan pass.

second, in cli_bytecode_prepare_jit, getPointerToFunction will be called on either on api functions or these wrapper functions of each bytecode entrypoint functions, this function will used to get the address of the specified function, compiling it if necessary. And as said before, this function will call function pass manger to run all registered pass to JIT the IR code.

```
// check the wrap functions, should be jited already
bcs->engine->compiledFunctions[func] = EE->getPointerToFunction(Functions[i]);

/// getPointerToFunction - This method is used to get the address of the specified function, compiling it if necessary.
void *JIT::getPointerToFunction(Function *F)
  getPointerToGlobalIfAvailable
  /// getPointerToGlobalIfAvailable - This returns the address of the specified
  /// global value if it is has already been codegen'd, otherwise it returns null.
```

eqmcc@http://blog.csdn.net/eqmcc

```
/// runJITOnFunction - Run the FunctionPassManager full of
/// just-in-time compilation passes on F, hopefully filling in
/// GlobalAddress[F] with the address of F's machine code.
runJITOnFunctionUnlocked
void JIT::runJITOnFunctionUnlocked(Function *F, const MutexGuard &locked)
   jitTheFunction(F, locked);
```

```
+ JIT function
void JIT::jitTheFunction(Function *F, const MutexGuard &locked)
   jitstate->getPM(locked).run(*F); <- run pass manager will also go following path
      bool FunctionPassManager::run(Function &F)
         // Execute all the passes managed by this top level manager.
         // Return true if any function is modified by a pass.
         bool FunctionPassManagerImpl::run(Function &F)
            for (unsigned Index = 0; Index < getNumContainedManagers(); ++Index)
             Changed |= getContainedManager(Index)->runOnFunction(F);
```

execute - run machine code for designated function call

TBD

cli_vm_execute_jit

```
// when called from clamav, func will be 0
// so entry point will always be called first as a start
void *code = bcs->engine->compiledFunctions[func];
// execute
ret = bytecode_execute((intptr_t)code, ctx);
```

bytecode_execute

```
ScopedExceptionHandler handler;
     // real execute;
```

eqmcc@http://blog.csdn.net/eqmcc

```
HANDLER_TRY(handler) {
// setup exception handler to longjmp back here
      uint32_t result = ((uint32_t (*)(struct cli_bc_ctx *))(intptr_t)code)(ctx);
      *(uint32_t*)ctx->values = result;
      return 0; // success and return
}
HANDLER_END(handler);

// a failure
return CL_EBYTECODE;
```

eqmcc@http://blog.csdn.net/eqmcc