C++11 metaprogramming applied to software obfuscation

Black Hat Europe 2014 - Amsterdam

Sebastien Andrivet - sebastien@andrivet.com, @AndrivetSeb
Senior Security Engineer, SCRT Information Security, www.scrt.ch
CTO, ADVTOOLS SARL, www.advtools.com

Abstract. The C++ language and its siblings like C and Objective-C are ones of the most used languages[1]. Significant portions of operating systems like Windows, Linux, Mac OS X, iOS and Android are written in C and C++. There is however a fact that is little known about C++: it contains a Turing-complete sub-language executed at compile time. It is called C++ template metaprogramming (not to be confounded with the C preprocessor and macros) and is close to functional programming.

This white paper will show how to use this language to generate, at compile time, obfuscated code without using any external tool and without modifying the compiler. The technics presented rely only on C++11, as standardized by ISO[2]. It will also show how to introduce some form of randomness to generate polymorphic code and it will give some concrete examples like the encryption of strings literals.

Keywords: software obfuscation, security, encryption, C++11, metaprogramming, templates.

Introduction

In the past few years, we have seen the comeback of heavy clients and of client-server model. This is in particular true for mobile applications. It is also the return of off-line modes of operation with Internet access that is not always reliable and fast. On the other hand, we are far more concerned about privacy and security than in the old times and mobiles phones or tablets are easier to steal or to loose than desktops or laptops. We have to protect secrets locally. In some cases, we also need to protect intellectual property (for example when using DRM systems) knowing that we are giving a lot of information to the attacker, in particular a lot of binary code. This is different from the web application model where critical portions of code are executed exclusively on the server, behind firewalls and IDS/IPS (at least until HMTL5).

We have thus to protect software in a hostile environment and obfuscation is one of the tools available to achieve this goal, even if it is far from a bullet-proof solution. Popular software such as Skype is using obfuscation like the majority of DRM (Digital Rights Management) systems and several viruses (to slow down their study).

Obfuscation

Obfuscation is “the deliberate act of creating […] code that is difficult for humans to understand”[3]. Obfuscated code has the same or almost the same semantics than the original and obfuscation is transparent for the system executing the application and for the users of this application.

Barak and al.[4] introduced in 2001 a more formal and theoretical study of obfuscation: an obfuscator $\mathcal{O}$ is a function that takes as input a program P and outputs another program $\mathcal{O}$(P) satisfying the following two conditions:

- (functionality) $\mathcal{O}$(P) computes the same function as P.
- ("virtual black box" property) "Anything that can be efficiently computed from $\mathcal{O}$(P) can be efficiently computed given oracle access to P."

Their main result is that general obfuscation is impossible even under weak formalization of the above conditions.

This result puts limits of what we can expect from an obfuscator. In the remaining of our discussion, we will focus on obfuscators not as an universal solution but as a way to slow down reverse engineering of softwares. We will also focus on areas typically exploited by attackers. In other terms, we will follow a pragmatical approach, not a theoretical one. For a more theoretical presentation, see for example the thesis of Jan CAPPAERT[5].

Types of obfuscators

It is possible to classify obfuscators in several ways depending on assumptions and intents. A possible classification is the following[6]:

- Source code obfuscators: transformation of the source code of the application before compilation.
- Binary code obfuscators: transformation of the binary code of the application after compilation.

This classification mimics the traditional phases of compilation: front-end (dependent of the source language) and back-end (independent of the source language, dependent on the target machine)[7].

Source code obfuscators can be further refined:

- Direct source code obfuscation: manual transformation of the source code by a programmer to make it difficult to follow and understand (including for other developers or for himself).
- Pre-processing obfuscators: automatic transformation of source code into modified source code before compilation.
- Abstract syntax tree (AST) or Intermediate representation (IR) obfuscators: compilers operate in phases. Some are generating an intermediate representation, a kind of assembly language or virtual machine bytecode (as it is the case for LLVM). This class of obfuscators transforms this intermediate language.
- Bytecode obfuscators: transformation of bytecode generated by the compiler (Java, .NET languages, etc.) It is a special case and share similarities with Abstract syntax tree obfuscators. This class of obfuscators is in fact located between source code and binary code obfuscators. We classify it in source code obfuscators because it is dependent of the languages and not of the target machine.

Under some circumstances, software or portion of it has to be released in source code. A typical example is javascript embedded in web pages. In this case, only some source code obfuscators are applicable.

C++

Depending on the language, it is possible to further refine this classification or to add new classes of obfuscators. It is the case for the C++ language[8]. Beyond the classical syntax and lexical analysis, C++ compilers incorporate other compilation phases: the pre-processor is well-known as it is directly inherited (almost without modifications) from the C language[9]. But there is another one, specific to C++: templates instantiation. It is this mechanism that will be used for the obfuscator described in this document.

C++11 template metaprogramming

Before going into the description of our obfuscator, it is necessary to give some basis of the mechanism involved: C++ template metaprogramming.

Templates

Originally, templates were designed to enable generic programming and provide type safety. A classical example is the design of a class representing a stack of objects. Without templates, the stack will contain a set of generic pointers without type information (i.e. of void*). As a consequence, it is possible to mix incompatible types and it is required to cast (explicitly or implicitly) pointers to appropriate types. The compiler is not able to enforce consistency. This is delegated to the programmer.

With templates, the situation is different: it is possible to declare and use a stack of a given type and the compiler will enforce it and produce a compilation error in case of a mismatch:

```
template<typename T>
struct Stack
{
    void push(T* object);
    T* pop();
};

Stack<Singer> stack;
stack.push(new Apple());    // compilation error
```

Contrary to other languages like Java, such templates do retain the types of objects they are manipulating. Each instance of a template generates code for the actual types used. As a consequence, the compiler has more latitude to optimize generated code by taking into account the exact context. Moreover, and thanks to a mechanism called specialization, this kind of optimization is also accessible to the programmer. For example, it is possible to declare a generic *Vector* template for objects and another version specialized for boolean. The two templates share a common interface but can use a completely different internal representation.

```
// Generic Vector for any type T
template<typename T>
struct Vector
{
    void set(int position, const T& object);
    const T& get(position);
```

```
    // ...
};

// template specialization for boolean
template<>
struct Stack<bool>
{
    void set(int position, bool b);
    bool get(position);
    // ...
};
```

Variadic templates

There are several situations where it is necessary to manipulate a list of types. It is the case for example when defining a tuple, a list of values of various types. Until C++11, the number of types (and thus of values) were arbitrarily limited by the implementation. It is not the case anymore with the latest versions of C++ (11 and 14): they are able to manipulate a list of types with variadic templates. For example, tuple can be defined by the following code:

```
template <typename... T>
class tuple {
public:
    constexpr tuple();
    explicit tuple(const T&…);
    […]
};
```

A tuple is created and used this way:

```
tuple<int, string, double> values{123, “test”, 3.14};
cout << get<0>(values);
```

Or, by using make_tuple helper:

```
auto values = make_tuple(123, “test”, 3.14);
cout << get<0>(values);
```

It is important to note that *make_tuple* and *get* are evaluated at compile time, not at runtime. They are compile-time entities.

Constexpr

This is another feature specific to C++11 and 14. It specifies that a value or function can be evaluated entirely at compile time (constant expression). As a consequence, only a subset of C++ is allowed. It was specifically added to the language for metaprogramming (see below). It implies both const and inline (in C++11). For example:

```
constexpr int factorial(int n)
{
    return n <= 1 ? 1 : (n * factorial(n-1));
}
```

Again, this is evaluated at compile time.

User-defined literals

In C++11 and 14, it is possible to define custom literals. For example, you can define a subset of the International System of Units and write[10]:

```
auto distance = 10_m;          // 10 meters
auto time = 20_s;              // 20 seconds
auto speed = distance / time; // 0.5 m/s
```

The compiler will check the consistency of expressions and will, for example, refuse:

```
if(speed == distance)   // compilation error
    [...]
```

These new suffixes are declared with code such as the following:

```
constexpr Quantity<M> operator"" _m(double d) { … }
```

We will try to use such custom suffix to declare obfuscated strings of characters but it will have limitations.

Metaprogramming

It was not the original intent of the designers of C++[11] but C++ templates is in fact a sub-language. This language is Turing-complete[12] and similar to functional programming. It is evaluated entirely at compile time, not at run time. For example, it is possible to declare the following:

```
template<int N>
struct Fibonacci { static constexpr int value = Fibonacci<N-1>::value +
Fibonacci<N-2>::value; };

template<>
struct Fibonacci<1> { static constexpr int value = 1; };

template<>
struct Fibonacci<0> { static constexpr int value = 0; }
```

It is an implementation of Fibonacci sequence[13] using recursion (note: it can be implemented differently, this code is designed this way to illustrate our discussion). The code:

```
Fibonacci<20>::value
```

is entirely computed at compile time and will be replaced by its result (6756). There is no computing and no cost at run time. We use recursion because C++ templates define a functional language: there is no variables, no loops, etc. Every statement is immutable like in Lisp[14] or Haskell[15].

Using this sub-language, we are able to generate code and not only to compute numbers. Templates are able to operate on types and make computation on them, or on other templates. We will use these possibilities to implement obfuscation schemas like encryption of string literals.

Encryption of strings literals

Strings literals are one of the most important source of information for an attacker when reverse engineering binaries. They are sometimes even more important than debugging information (when they are available). Thanks to those literals, the attacker will be able to quickly find interesting portion of code instead of trying to take a costly top-bottom approach (reverse engineering from the entry point of the binary). Binary often contains several different kind of string literals like:

- error messages
- log information (even if logs are not activated)
- name of functions or of classes
- URLs
- etc.

It is essential to obfuscate these literals in order to slow down reverse engineering. Some programmers obfuscate these literals manually (direct source code obfuscation) and maintain (manually) a list of correspondence between obfuscated strings and original ones. This kind of solution is difficult (if ever possible) to maintain. Others use a pre-processor to automate these modifications. But again, it is difficult to maintain and it makes debugging more difficult for the developer.

Our goal is to obfuscate string literals with the following constraints:

- use a developer-friendly syntax. In particular, the original string literal has to be present in source code.
- use only C++ without any external tool.
- obfuscate literals at compile time. De-obfuscation can be performed at runtime.
- the cost of obfuscation / deobfucation has to be minimal.
- the original string must not be present in the binary in release builds. It is acceptable if it is present in debug builds.

In a second phase, we will add the following constraints:

- each string literal has to be obfuscated differently.
- each compilation of the same source code has to produce different obfuscated strings.

First implementation

The first tentative of implementation is the following:

```
template<int... I>
struct MetaString1
{
    constexpr ALWAYS_INLINE MetaString1(const char* str)
    : buffer_ {encrypt(str[I])...} { }

    const char* decrypt()
    {
```

```
        for(int i = 0; i < sizeof...(I); ++i)
            buffer_[i] = decrypt(buffer_[i]);
        buffer_[sizeof...(I)] = 0;
        return buffer_;
    }

private:
    constexpr char encrypt(char c) const { return c ^ 0x55; }
    constexpr char decrypt(char c) const { return encrypt(c); }

private:
    char buffer_[sizeof...(I) + 1];
};

constexpr ALWAYS_INLINE const char* operator "" _obfuscated1(const char* str,
size_t)
{
    return MetaString1<0, 1, 2, 3, 4, 5>(str).decrypt();
}

#define OBFUSCATED1(str) (MetaString1<0, 1, 2, 3, 4, 5>(str).decrypt())
```

It defines a template class called *MetaString1*. Its constructor accepts a parameter: a string of characters (const char*). The class contains also a private buffer called *buffer_*. In the constructor, the buffer is initialized: each character (*str[I]*) is encrypted with the *encrypt* function. It assumes that the template parameter *I* will contain the suite of integers 0, 1, 2, … This is called a variadic template. This way, the buffer will be initialized with the encrypted characters *str[0], str[1], str[2]*, etc.

The encryption is very simple in this first version: it simply makes a XOR of the value of a character with the hard-coded value *55* (in hexadecimal). This is far from optimal and will be enhanced in a later version. Since we are using XOR, decryption is identical to encryption. To decrypt a string we simply iterate character decryption for each byte in the internal buffer. This is done in place (decrypted characters replace obfuscated ones).

In release builds and with the help of *always_inline* attribute, most compilers are able to *inline* all member functions. In other terms, the content of members is directly injected into the code of callers. There is no call to member functions.

The custom operator “” defines a new suffix *_obfuscated1* and allows writing:

```
cout << "Britney Spears"_obfuscated1 << endl;
```

The macro *OBFUSCATED1* can be used as an alternative:

```
cout << OBFUSCATED1("Britney Spears") << endl;
```

Version 2 - Generation of a list of integers to instantiate variadic templates

This first version is a good starting point. However, it has a major drawback: it is only able to handle strings of 6 or less characters. A list of indexes (0, 1, 2, 3, 4, 5) is hard-coded and passed explicitly to instantiate *MetaString1*. Of course, it is possible to add more indexes but it will still be limited.

C++14 standard library introduces *std::index_sequence* and related types to generate lists of integers. But as only a few compilers are implementing C++14 today, we rely on a custom and simplified implementation. The code:

```
MakeIndex<N>::type
```

will generate:

```
Indexes<0, 1, 2, 3, …, N>
```

Thanks to this helper, it is possible to remove the hardcoded list of indexes and generate it at compile time:

```
MetaString2<Make_Indexes<sizeof(str) - 1>::type>(str)
```

Sizeof gives the length of the literal with the terminating null byte. Unfortunately, it is not possible to use such technique with user-define suffixes: the length of the literal is passed as a parameter but it is not considered constant and thus it can't be used in metaprogramming. We have thus to use the macro:

```
cout << OBFUSCATED2("Katy Perry") << endl;
```

This time, the string is not truncated.

Version 3 - Randomization of encryption keys

In the previous version, the encryption key is hard-coded (0x55) and is thus the same for all strings. The third version introduces a random number generator evaluated at compile time. It uses a Lehmer random number generator[16]. It is certainly possible to use a better algorithm but it is simple to implement, well known and considered a minimal standard[17]. In our case, as we just want to obfuscate literals, we need random numbers but their randomness is not so important (after all, our keys have only 8 bits; it is easy to brute-force). What is important is to use a different key for each compilation and each string literal.

In order to use different keys for each compilation, it is essential to use an appropriate seed. Some authors[18] are suggesting to define externally (with a build script) a macro such as *__RANDOM__*. There is however a more simple solution: use the compilation time defined by *__TIME__*[19]. It can be converted to a compile time value with the following code:

```
// __TIME__ has the following format: hh:mm:ss in 24-hour time
constexpr char time[] = __TIME__;

constexpr int DigitToInt(char c) { return c - '0'; }
const int seed = DigitToInt(time[7]) +
                 DigitToInt(time[6]) * 10 +
                 DigitToInt(time[4]) * 60 +
                 DigitToInt(time[3]) * 600 +
                 DigitToInt(time[1]) * 3600 +
                 DigitToInt(time[0]) * 36000;
```

In order to generate a different pseudo-random number for each literal, we also use the macro *__COUNTER__*[20]. This later is not standard but defined by a majority of compilers. If is possible to use other macros such as *__LINE__* and *__FILE__* (they are standard) to achieve maximum portability but the implementation will be slightly more complex.

Version 4 - Randomization of encryption algorithm

It is possible to push the principle described in the previous point further: We generate a different key for each compilation and for each string literal. But we can also use a different encryption algorithm. We introduce a new template parameter (A) for *MetaString*:

```
template<int A, int Key, typename Indexes>
struct MetaString4;
```

We then define a partial specialization for each possible value of **A**:

```
template<int K, int... I>
struct MetaString4<0, K, Indexes<I...>>
{
    …
};


template<int K, int... I>
struct MetaString4<1, K, Indexes<I...>>
{
    …
};
```

Each specialization uses a different encryption algorithm. In our example:

- The first one (A = 0) makes an XOR of each character with a random key.
- The second one (A = 1) makes also an XOR but the value of the key is incremented for each character.
- The third one (A = 2) shifts the value of each character by a random value (variation of the caesar cipher).

These are only simple examples to illustrate the principle. It is possible to use far more complex implementations (and for example use a wider key length) but it may have an impact on performance during runtime. In particular, using AES seems excessive in this particular case.

As for the key, we select the algorithm with an expression such as:

```
MetaRandom<__COUNTER__, 3>::value
```

Generated binary code

The disassembly of the following code (without obfuscation):

```
int main(int argc, const char * argv[])
{
    cout << "Britney Spears";
    return 0;
}
```

gives (Apple LLVM 5.1 / LLVM 3.4):

```
_main           proc near
                push    rbp
                mov     rbp, rsp
                mov     rdi, cs:__ZNSt3__14coutE_ptr
                lea     rsi, aBritneySpears ; "Britney Spears"
                call    __ZNSt3__1lsINS_11char_traitsIcEEEERNS_13
                xor     eax, eax
                pop     rbp
                retn
_main           endp
```

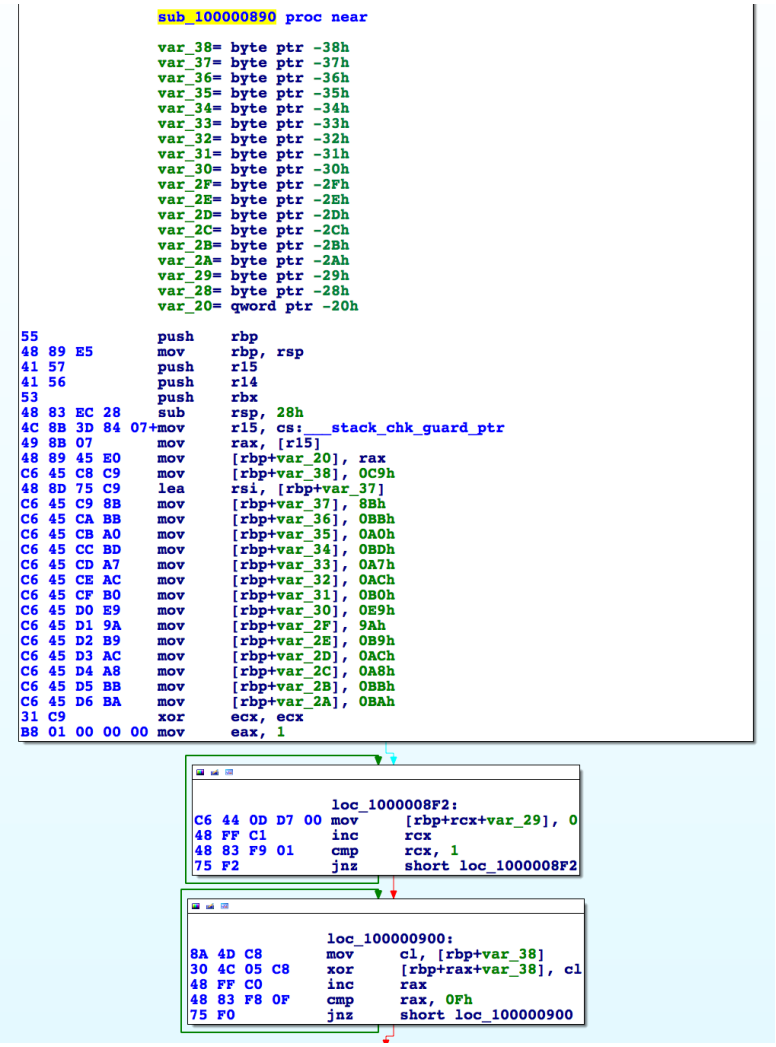
The string literal is in plain text:

Address	Length	Type	String
HEADER:0000000100000504	0000000E	C	/usr/lib/dyld
HEADER:0000000100000580	00000018	C	/usr/lib/libc++.1.dylib
HEADER:00000001000005B0	0000001B	C	/usr/lib/libSystem.B.dylib
__cstring:0000000100000F4C	0000000F	C	Britney Spears
__eh_frame:0000000100000FE9	00000005	C	zPLR

The equivalent code using the obfuscator:

```
int main(int argc, const char * argv[])
{
    cout << OBFUSCATED4("Britney Spears") << endl;
    return 0;
}
```

gives (Apple LLVM 5.1 / LLVM 3.4):



The first block allocates space on the stack and stores the value of each encrypted character. These values are mixed with machine code (C6 45 - mov). The third block decrypts the string (in this case, it is a XOR with the key loaded from var_38 and initialized with 0xC9 in the first block). There is no call: the compiler has inlined all methods.

The binary does not contain the original string and the encrypted one does not appear either:

Address	Length	Type	String
HEADER:0000000100000554	0000000E	C	/usr/lib/dyld
HEADER:00000001000005D0	00000018	C	/usr/lib/libc++.1.dylib
HEADER:0000000100000600	0000001B	C	/usr/lib/libSystem.B.dylib
__eh_frame:0000000100000FE1	00000005	C	zPLR

Obfuscation using Finite State Machines

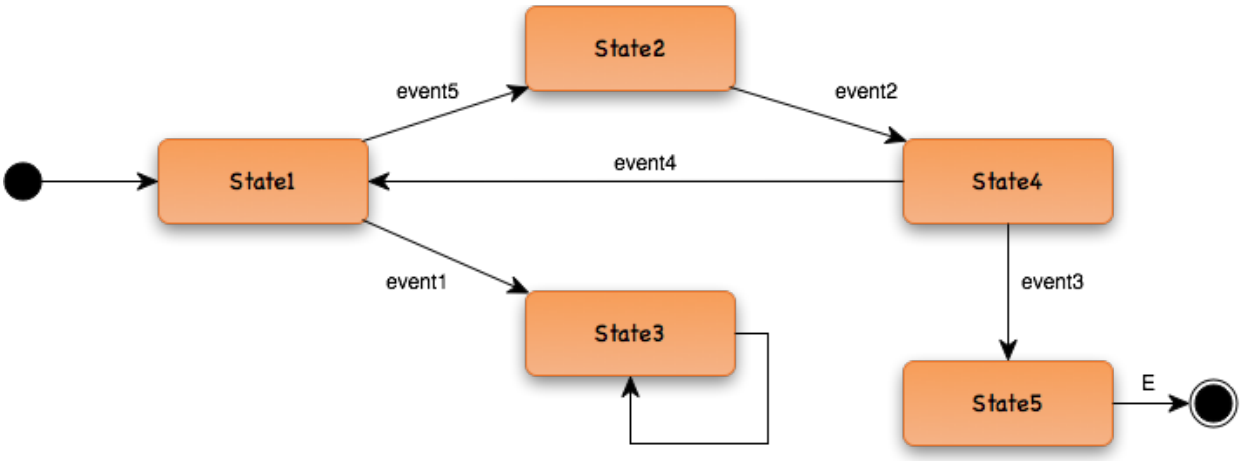
Using the same techniques than those described previously, it is possible to obfuscated other areas of an application. As an illustration, the obfuscator is able to obfuscate calls. Code such as:

```
function_to_protect();
```

is obfuscated by rewriting it as:

```
OBFUSCATED_CALL(function_to_protect);
```

The obfuscator will instantiate a finite state machine such as (simple example for illustration):



Before actually call *function_to_protect*, this finite state machine will run until it reaches the final state. Such schema will slow down both static and dynamic analysis.

The finite state machine is based on Boost Meta State Machine (MSM) library[21]. Boost[22] is a set of free and peer-reviewed C++ libraries. They are designed to work well with the C++ Standard Library. Some of the libraries of Boost were integrated in the Standard Library (in TR1 and C++ 11 in particular).

Boost Meta State Machine allows to easily and quickly define state machines with very high performance"[23]. Without going too much into details, the central entity of MSM is a transition table defined this way:

```
struct transition_table : mpl::vector<
//    Start     Event         Next      Action               Guard
//  +---------+-------------+---------+--------------------+----------------------+
Row < State1  , event5      , State2                                               >,
Row < State1  , event1      , State3                                               >,
//  +---------+-------------+---------+--------------------+----------------------+
Row < State2  , event2      , State4                                               >,
//  +---------+-------------+---------+--------------------+----------------------+
Row < State3  , none        , State3                                               >,
//  +---------+-------------+---------+--------------------+----------------------+
Row < State4  , event4      , State1                                               >,
Row < State4  , event3      , State5                                               >,
//  +---------+-------------+---------+--------------------+----------------------+
Row < State5  , E           , Final,    CallTarget                                 >
//  +---------+-------------+---------+--------------------+----------------------+
> {};
```

This a compile-time vector: this table is not instantiated at runtime but it is used at compile time to generate the finite state machine code. *State1*, *event1*, … are not values but arbitrary types representing states and events.

The address of the function to call is also obfuscated with the same techniques than what we have used for the obfuscation of string literals. Otherwise, tools such as IDA[24] will be able to computer references and find callers and callees.

Without obfuscation, the following code:

```
function_to_protect();

int result = function_to_protect_with_parameter("did", “again");
```

looks like (Apple LLVM 5.1 / LLVM 3.4):

```
                sub_10000160E   proc near
55                              push    rbp
48 89 E5                        mov     rbp, rsp
E8 E9 FD FF FF                  call    sub_100001400
48 8D 3D 49 66+                 lea     rdi, aDid       ; "did"
48 8D 35 46 66+                 lea     rsi, aAgain     ; "again"
5D                              pop     rbp
E9 C7 FE FF FF                  jmp     sub_1000014F2
                sub_10000160E   endp
```

With obfuscation, the corresponding code:

```
OBFUSCATED_CALL(function_to_protect);

int result = OBFUSCATED_CALL_RET(int, function_to_protect_with_parameter,
OBFUSCATED4("did"), OBFUSCATED4(“again”));
```

looks like (this is only a small subset):

```
48 C7 85 44 FF+                 mov     [rbp+var_BC], 0
48 C7 85 3C FF+                 mov     [rbp+var_C4], 0
48 8D BD E8 FE+                 lea     rdi, [rbp+var_118]
48 8D B5 38 FF+                 lea     rsi, [rbp+var_C8]
E8 49 43 00 00                  call    sub_100005A2A
C7 85 EC FE FF+                 mov     [rbp+var_114], 0
48 8D BD E8 FE+                 lea     rdi, [rbp+var_118]
48 8D 75 D0                     lea     rsi, [rbp+var_30]
BA 01 00 00 00                  mov     edx, 1
E8 2E 46 00 00                  call    sub_100005D2E
BB 45 00 00 00                  mov     ebx, 45h
4C 8D AD E8 FE+                 lea     r13, [rbp+var_118]
4C 8D 75 C8                     lea     r14, [rbp+var_38]
4C 8D 7D C0                     lea     r15, [rbp+var_40]
4C 8D 65 B8                     lea     r12, [rbp+var_48]

                loc_100001718:                          ; CODE XREF: sub_10000163A+101 j
4C 89 EF                        mov     rdi, r13
4C 89 F6                        mov     rsi, r14
E8 2F 0C 00 00                  call    sub_100002352
4C 89 EF                        mov     rdi, r13
4C 89 FE                        mov     rsi, r15
E8 40 0D 00 00                  call    sub_10000246E
4C 89 EF                        mov     rdi, r13
4C 89 E6                        mov     rsi, r12
E8 51 0E 00 00                  call    sub_10000258A
FF CB                           dec     ebx
75 DB                           jnz     short loc_100001718
48 8D BD E8 FE+                 lea     rdi, [rbp+var_118]
48 8D 75 B0                     lea     rsi, [rbp+var_50]
E8 05 0C 00 00                  call    sub_100002352
48 8D BD E8 FE+                 lea     rdi, [rbp+var_118]
48 8D 75 A8                     lea     rsi, [rbp+var_58]
E8 11 0D 00 00                  call    sub_10000246E
48 8D BD E8 FE+                 lea     rdi, [rbp+var_118]
48 8D 75 A0                     lea     rsi, [rbp+var_60]
E8 39 0F 00 00                  call    sub_1000026A6
48 8D 05 64 FE+                 lea     rax, loc_1000015D7+1
48 89 45 88                     mov     [rbp+var_78], rax

                loc_100001778:                          ; DATA XREF: sub_10000163A+332 o
C7 45 90 B8 01+                 mov     [rbp+var_70], 1B8h
48 8D BD E8 FE+                 lea     rdi, [rbp+var_118]
48 8D 75 88                     lea     rsi, [rbp+var_78]
BA 01 00 00 00                  mov     edx, 1
E8 42 43 00 00                  call    sub_100005AD6
48 8D BD F0 FE+                 lea     rdi, [rbp+var_110]
E8 D4 15 00 00                  call    sub_100002D74
48 8D BD 40 FF+                 lea     rdi, [rbp+var_C4+4]
E8 C8 15 00 00                  call    sub_100002D74
C7 85 D8 FD FF+                 mov     [rbp+var_228], 0F020F6Bh
31 C9                           xor     ecx, ecx
B8 01 00 00 00                  mov     eax, 1

                loc_1000017BD:                          ; CODE XREF: sub_10000163A+192 j
C6 84 0D DC FD+                 mov     [rbp+rcx+var_224], 0
48 FF C1                        inc     rcx
48 83 F9 01                     cmp     rcx, 1
75 EF                           jnz     short loc_1000017BD
```

None of the addresses loaded (LEA) or called (CALL) are the actual addresses of our functions *function_to_protect* and *function_to_protect_with_parameter*. It will thus slow down reverse engineering by attackers.

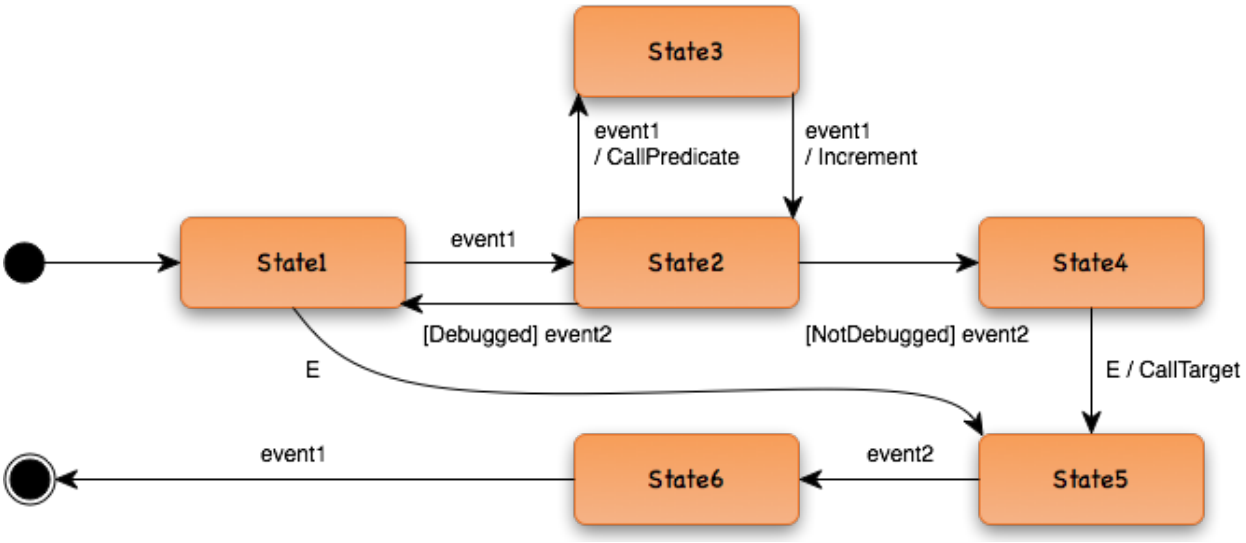
Random selection of Finite State Machine

We see previously how to select an encryption algorithm for string literals. Using exactly the same technique, it is possible to randomly select a finite state machine from a set. It is also possible to randomly change some part of the implementation, such as the obfuscation of function addresses. It is transparent for the user of the obfuscator. The only constraint is to use viable state machines (i.e. machines with a path to the final state).

Combining with anti-debug & anti-VM measures

We can also combine state transitions with debugger or virtual machine detection: depending on the result of the detection, the machine will follow other paths of execution and eventually crash or enter an infinite loop.

For example, the companion code contains the following finite state machine:



It is represented by the following compile-time structure (meta-vector):

```
struct transition_table : mpl::vector<
//    Start     Event         Next      Action                 Guard
//  +---------+-------------+---------+----------------------+----------------------+
Row < State1  , event1      , State2                                                 >,
Row < State1  , E           , State5                                                 >,
//  +---------+-------------+---------+----------------------+----------------------+
Row < State2  , event1      , State3  , CallPredicate                                >,
Row < State2  , event2      , State1  , none                 , Debugged              >,
Row < State2  , event2      , State4  , none                 , NotDebugged           >,
//  +---------+-------------+---------+----------------------+----------------------+
Row < State3  , event1      , State2  , Increment                                    >,
//  +---------+-------------+---------+----------------------+----------------------+
Row < State4  , E           , State5  , CallTarget                                   >,
//  +---------+-------------+---------+----------------------+----------------------+
Row < State5  , event2      , State6                                                 >,
//  +---------+-------------+---------+----------------------+----------------------+
Row < State6  , event1      , Final                                                  >
//  +---------+-------------+---------+----------------------+----------------------+
> {};
```

For *State2*, the machine can follow two different paths for the same event: *event2*. The difference is the guard condition: the path on the left is conditioned by the predicate *Debugged* and the path on the right, by the predicate *NotDebugged*. *Debugged* and *NotDebugged* are not exactly function. They are functors, classes that mimic functions my implemented the call operator (operator()):

```
struct NotDebugged
{
    template<typename EVT, typename FSM, typename SRC, typename TGT>
    bool operator()(EVT const& evt, FSM& fsm, SRC& src, TGT& tgt)
    {
        return !Debugged{}(evt, fsm, src, tgt);
    }
};
```

The implementation of *NotDebugged* is simple: it is simply the contrary of *Debugged*.

The implementation of *Debugged* is more subtle. It possible to make some tests, use a “if” instruction and return a boolean value. As an example of what is possible, I choose another implementation: the presence of a debugger is tested when *State3* is reached (*CallPredication* action) and the result increment a counter. The counter is also incremented when the FSM switch from *State3* to *State2*. This is done a certain number of time, determined randomly (at compile time). The result is that the counter is even if a debugger was detected and odd otherwise. The idea is to separate in time the actual detection from the usage of this detection. *Debugged* is thus implemented as:

```
struct Debugged
{
    template<typename EVT, typename FSM, typename SRC, typename TGT>
    bool operator()(EVT const& evt, FSM& fsm, SRC& src, TGT& tgt)
    {
        return (fsm.predicateCounter_ - fsm.predicateCounterInit_) % 2 == 0;
    }
};
```

The companion code contains a working implementation of debugger detection for Mac OS X and iOS. It is a simple implementation (based on a document from Apple). A more realistic implementation would incorporate obfuscation techniques described in this white paper to make the implementation more difficult to recognize and remove. For example, it is possible to use another FSM machine to hide the calls to *sysctl* and *getpid*. Another possibility is to make this function inline, call it from different part of the FSM, and use more complex mathematics that just increments and even testing.

To make the code more generic, the companion code does not define directly Debugged and NotDebugged. Instead, it defines generic Predicate and NotPredicate functors. The actual implementation of the predicate is a template parameter when calling the function to obfuscate:

```
// Predicate
struct DetectDebugger { bool operator()() { return AmIBeingDebugged(); } };

void SampleFiniteStateMachine2()
{
   OBFUSCATED_CALL_P(DetectDebugger,
       SampleFiniteStateMachine_important_function_in_the_application);
}
```

In this example, *SampleFiniteStateMachine_important_function_in_the_application* is only called if *AmIBeingDebugged* return false. The whole mechanism is obfuscated by the FSM.

Other areas and future directions

The same principles and techniques are applicable to other areas such as the obfuscation of computations, the introduction of opaque predicates, etc.

Mixing with Objective-C

Within Apple Xcode, It is possible to mix C, Objective-C and C++ in the same project or even in the same file: Xcode supports what is called Objective-C++ with extension ".mm" (instead of ".m"). This way, it is possible to use the techniques described in this document within iOS and Mac OS X applications.

A characteristic of Objective-C is that all calls are dynamic and use what is called "selectors". To simplify, a selector is the name of a method and this name is preserved in compiled code. As a consequence, it gives valuable information to attackers. Currently, our obfuscator is not addressing this area but this is currently studied and may be part of an update.

With the release of Swift, the interest to obfuscate selectors has shift from Objective-C to this new language.

Compilers support

This library was developed using Xcode 6.0 and 6.1 beta. The corresponding LLVM version is 3.5. It will however compile and run with any C++11 or C++1y (14) conforming compiler.

It is currently **not** compatible with Microsoft Visual C++ including update 3 of Visual Studio 2013 and Visual Studio 2014 CTP3. The main reason is the lack of support of *constexpr* and of initialisation of arrays. They are only partially supported by Microsoft. Currently, it is not clear if the final release of Visual C++ 14 will fully support *constexpr* or not[25].

The following table summarizes compatibility:

Compilers	Compatibility	Remarks
Apple LLVM 5.1 (3.4)	**Yes**	Previous versions were not tested
Apple LLVM 6.0 (3.5)	**Yes**	Xcode 6, 6.1 beta
LLVM 3.4, 3.5	**Yes**	Previous versions were not tested
GCC 4.8.2 and higher	**Yes**	Previous versions were not tested
Intel C++ 2013	**Yes**	Version 14.0.3 (2013 SP1 Update 3)
Visual Studio 2013 U3	No	Lack of constexpr support
Visual Studio 2014 TP	Almost	Lack of initialisation of arrays support
Visual Studio 2014 RTM	Unknown	Not yet released at the time of this writing

Side effects and performance

The impact at compile time and at runtime of obfuscation techniques is largely dependent of the context. For example, if you design a big finite state machine or if you make several iterations

during its execution, it will slow down the application. This is why in our example MetaString, we use simple operations like XORs.

As a general guideline, it is better to protect only specific portions of code. As an example, the obfuscator presented here was originally created to protect jailbreak detection code in an iOS framework. Protecting other areas such as all user interface code is more questionable.

Comparison with other obfuscators

There are only a few obfuscators available. Some are commercial like Arxan, Metaforic, Morpher or Cryptanium. Only very few are open-source. This is the case of Obfuscator-LLVM[26] (they are a few others but they are more proofs of concept than actual products).

They all rely on external tools (pre-processors, post-processors, modified versions of LLVM, profilers, …) to produce obfuscated binaries. Our approach is different and relies only on C++11. Each approach has its benefits and drawbacks. Both are not incompatible and can be combined to further obfuscate binaries.

The following table summarizes benefits and drawbacks of our approach:

Benefits	Drawbacks
Does not rely on external tools or modified version of the compiler	The C++ compiler has to be C++11 compliant
Not dependent on the target platform (the target has to be supported by a C++11 compiler)	Some part of the source code has to be in C, C++ or Objective-C
Very few impact on the source code (only a little intrusive)	Complex to write and to debug
Obfuscate at high-level. Allows complex obfuscation involving different parts of an application	Some obfuscation techniques like control flow graph flattening seem more difficult to implement without an important impact on the source code of the application
Our approach is applicable in environments where it is forbidden to dynamically decrypt or decode binary code (such as Apple iOS)	

Companion code

A version of our obfuscator is available on GitHub (https://github.com/andrivet/ADVobfuscator). It contains examples of techniques such as:

- Obfuscation of string literals with random keys and random encryption algorithm
- Obfuscation of function call with finite state machine
- Obfuscation of function call mixed with a predicate (debugger detection for Mac OS X and iOS)

The repository contains a Xcode 6.0 project that generates a Mac OS X Command Line tool. It demonstrates each point explained in the present document including intermediate steps:

File	Description
Indexes.h	Generate list of indexes at compile time (0, 1, 2, … N)
MetaFactorial.h	Compute factorial at compile time
MetaFibonacci.h	Compute fibonacci sequence at compile time
MetaRandom.h	Generate a pseudo-random number at compile time
MetaString1.h	Obfuscated string - version 1
MetaString2.h	Obfuscated string - version 2 - Remove truncation
MetaString3.h	Obfuscated string - version 3 - Random key
MetaString4.h	Obfuscated string - version 4 - Random encryption algorithm
ObfuscatedCall.h	Obfuscate function call
main.cpp	Samples

Conclusion

This document and its companion code demonstrate that it is possible to use C++11 compilers to obfuscate code without using any external tools or modifying the compiler. For example, our obfuscator is able to obfuscate string literals and function calls. Such techniques can be extended to obfuscate code further by using identities, opaque predicates, etc.

The techniques described in this document were successfully applied in products, including commercial ones. In particular, it was used to protect jailbreak detection code in iOS applications published on the AppStore.

We are continuing our researches, in particular regarding the obfuscation of code written in Swift. We are also looking for solutions to apply similar techniques to Android code written in Java.

History

Version 0	December 1, 2011	First version, strings literals obfuscation, experimental
Version 1	May 25, 2013	Major enhancements, based on work from Samuel Neves, Filipe Araujo and on work from malware maker “LeFF”. Applied to ADVdetector (commercial product)
Version 2	June 7, 2014	Enhancements for Hack In Paris 2014. Choose obfuscation algorithm randomly, experiments with finite state machines
Version 3	September 26, 2014	Enhancements for Black Hat Europe 2014. Choose finite state machine (FSM) randomly from a set, change FSM behavior depending on a runtime value (debugger detection)

To get the latest version of this document, please visit:

https://github.com/andrivet/ADVobfuscator/tree/master/Docs

References

[1] TIOBE Index for May 2014, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] ISO/IEC 14882:2011 - ANSI eStandards Store, http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2fISO%2fIEC+14882-2012

[3] Wikipedia, May 2014 - http://en.wikipedia.org/wiki/Obfuscation_(software)

[4] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001) - http://www.iacr.org/archive/crypto2001/21390001.pdf

[5] Jan CAPPAERT, Arenberg Doctoral School of Science, Engineering & Technology

[6] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere: On the Effectiveness of Source Code Transformations for Binary Obfuscation

[7] Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman: Compilers Principles, Technics and Tools, Addison-Wesley (1986)

[8] ISO/IEC 14882:2011, January 2012 Draft, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[9] ISO/IEC JTC1/SC22/WG14; C99 ISO/IEC 9899:1999; C11 ISO/IEC 9899:2011

[10] Bjarne Stroustrup, The C++ Programming Language, Fourth Edition, page 822, http://www.stroustrup.com/4th.html

[11] Prime numbers in error messages, http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s04.html#Static-metaprogramming

[12] http://en.wikipedia.org/wiki/Turing-complete

[13] http://en.wikipedia.org/wiki/Fibonacci_number

[14] Common Lip, http://common-lisp.net

[15] http://www.haskell.org

[16] W.H. Payne, J.R. Rabung, T.P. Bogyo (1969). "Coding the Lehmer pseudo-random number generator

[17] Stephen K. Park and Keith W. Miller (1988): Random Number Generators: Good Ones Are Hard To Find

[18] Samuel Neves, Filipe Araujo (2012): Binary code obfuscation through C++ template metaprogramming

[19] C and C++ Syntax Reference, Cprogramming,com, __TIME__, http://www.cprogramming.com/reference/preprocessor/__TIME__.html

[20] Predefined Macros, Microsoft, http://msdn.microsoft.com/en-us/library/b0084kay.aspx

[21] http://www.boost.org/doc/libs/1_55_0/libs/msm/doc/HTML/index.html

[22] http://www.boost.org

[23] http://www.boost.org/doc/libs/1_55_0/libs/msm/doc/HTML/pr01.html

[24] https://www.hex-rays.com/products/ida/

[25] VC++ Conformance Update, http://blogs.msdn.com/b/somasegar/archive/2014/05/28/first-preview-of-visual-studio-quot-14-quot-available-now.aspx

[26] Obfuscator-LLVM, University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains, http://www.o-llvm.org/