C++11 METAPROGRAMMING APPLIED TO SOFTWARE OBFUSCATION

SEBASTIEN ANDRIVET
About me

Sebastien ANDRIVET
Cyberfeminist & hacktivist
Reverse engineer Intel & ARM
C++, C, Obj-C, C# developer
Trainer (iOS & Android appsec)

Senior Security Engineer
at SCRT (Swiss)

CTO
at ADVTOOLS (Swiss)
PROBLEM
Reverse engineering

- Reverse engineering of an application if often like following the “white rabbit”
  - i.e. following string literals

- Live demo
  - Reverse engineering of an application using IDA
  - Well-known MDM (Mobile Device Management) for iOS
A SOLUTION

OBfuscation
What is Obfuscation?
Obfuscator $O$
YES! It is also Katy Perry!

- (almost) same semantics
- obfuscated
Obfuscation

“Deliberate act of creating source or machine code difficult for humans to understand”

—WIKIPEDIA, APRIL 2014
C++ templates

- Example: Stack of objects
  - Push
  - Pop
Without templates

```cpp
class Stack {
  void push(void* object);
  void* pop();
};

Stack singers;
singers.push(britney);

Stack apples;
apples.push(macintosh);
```

- Reuse the same code (binary)
- Only 1 instance of Stack class
With C++ templates

template<typename T>
class Stack
{
    void push(T object);
    T pop();
};

Stack<Singer> singers;
singers.push(britney);

Stack<Apple> apples;
apples.push(macintosh);
With C++ templates

Stack<Singer> singers;
singers.push(britney);

Stack<Apple> apples;
apples.push(macintosh);
C++ templates

- Two instances of Stack class
  - One per type
- Does not reuse code
  - By default
- Permit optimisations based on types
  - For ex. reuse code for all pointers to objects
- Type safety, verified at compile time
Type safety

- `singers.push(apple);`  // compilation error
Optimisation based on types

- Generate different code based on types (template parameters)

```cpp
template<typename T>
class MyClass
{
...
    enable_if_t<is_pointer<T>::value, T>
    member_function(T t) { ... };
    ...
};
```

- Example: `enable_if`

- `member_function` is only defined if `T` is a pointer type

(Warning: C++14 code, not C++11)
C++ metaprogramming

- Programs that manipulate or produce programs
- Subset of C++
- Turing-complete (~ full programming language)
- Close to Functional programming
- Part of C++ standards
  - Major enhancements in C++11 et C++14
Application 1 - Strings literals obfuscation

- original string is source code
- original string in DEBUG builds
- developer-friendly syntax
- no trace of original string in compiled code in RELEASE builds
1st implementation

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

    const char* decrypt();

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

private:
    char buffer_[sizeof...(Indexes) + 1];
};
1st implementation

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

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

private:
    char buffer_[sizeof...(Indexes) + 1];
};
```
1st implementation

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

  const char* decrypt();
}

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

private:
  char buffer_[sizeof...(Indexes) + 1];
1st implementation - Usage

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

cout << OBFUSCATED1("Britney Spears") << endl;
1st implementation - Problem

- List of indexes is hard-coded
  - 0, 1, 2, 3, 4, 5
- As a consequence, strings are truncated!
2\textsuperscript{nd} implementation

- Generate a list of indexes with metaprogramming
- C++14 introduces \texttt{std:index\_sequence}
- With C++11, we have to implement our own version
  - Very simplified
  - \texttt{MakeIndex<N>::type} generates:
    - \texttt{Indexes<0, 1, 2, 3, ..., N>
2\textsuperscript{nd} implementation

• Instead of:

\texttt{MetaString1<0, 1, 2, 3, 4, 5>(str)}

• we have:

\texttt{MetaString2<Make\_Indexes<sizeof(str)-1>::type>(str)}
2\textsuperscript{nd} implementation - Usage

c\textbf{ou} << \texttt{OBFU}SC\texttt{A}TED2("Katy Perry") << endl;

- No more truncation
3rd implementation

- In previous implementations, key is hard-coded
  
  ```cpp
  constexpr char encrypt(char c) const { return c ^ 0x55; }
  ```

- New template parameter for Key
  
  ```cpp
  template<int... I, int K>
  struct MetaString3<Indexes<I...>, K>
  ```
Generating (pseudo-) random numbers

- C++11 includes `<random>`, but for runtime, not compile time

- **MetaRandom<N, M>**
  - **N**: Nth generated number
  - **M**: Maximum value (excluded)

- Linear congruential engine
  - Park-Miller (1988), "minimal standard"

- Not exactly a uniform distribution (modulo operation)

- Recursive
Seed

- template<>
  struct MetaRandomGenerator<0> {
    static const int value = seed;
  };

- How to choose an acceptable compile-time seed?

- Macros (C & C++):
  - __TIME__: compilation time (standard)
  - __COUNTER__: incremented each time it is used (non-standard but well supported by compilers)
3rd implementation

• Different keys for each compilation
  • thanks to __TIME__

• Different key for each string
  • thanks to __COUNTER__
4<sup>th</sup> implementation

- Different and random keys, great!
- Why not go even further?
- Choose a different encryption algorithm, randomly!
4\textsuperscript{th} implementation

- Template partial specialization

- \texttt{template\:<\texttt{int A}, int Key, typename Indexes> struct MetaString4;}

- \texttt{template<int K, int... I> struct MetaString4<0, K, Indexes<I...>> {};}

- \texttt{template<int K, int... I> struct MetaString4<1, K, Indexes<I...>> {};}

- \#define DEF_OBFUSCATED4(str) MetaString4<MetaRandom<\_\_COUNTER\_\_, 2>::value, ...}
Result

- **Without obfuscation**
  ```
  cout << "Britney Spears" << endl;
  ```

- **With obfuscation**
  ```
  cout << OBFUSCATED4("Britney Spears") << endl;
  ```
Without obfuscation

```assembly
_main
 proc near
 push    rbp
 mov     rbp, rsp
 mov     rdi, cs:__ZNSt3__14coutE_ptr
 lea     rsi, aBritneySpears ; "Britney Spears"
call __ZNSt3__11sINS_11char_traitsIcEEEERNS_1:
 xor     eax, eax
 pop     rbp
 ret
 endp
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Length</th>
<th>Type</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADER:0000000100000504</td>
<td>0000000E</td>
<td>C</td>
<td>/usr/lib/dyld</td>
</tr>
<tr>
<td>HEADER:0000000100000580</td>
<td>00000018</td>
<td>C</td>
<td>/usr/lib/libc++.1.dylib</td>
</tr>
<tr>
<td>HEADER:00000001000005B0</td>
<td>0000001B</td>
<td>C</td>
<td>/usr/lib/libSystem.B.dylib</td>
</tr>
<tr>
<td>__cstring:0000000100000F4C</td>
<td>0000000F</td>
<td>C</td>
<td>Britney Spears</td>
</tr>
<tr>
<td>__eh_frame:0000000100000FE9</td>
<td>00000005</td>
<td>C</td>
<td>zPLR</td>
</tr>
</tbody>
</table>
With obfuscation

Encrypted characters (mixed with MOV)

Decryption
Application 2 - Obfuscate calls

• How to obfuscate call such as:
  
  • `function_to_protect();`

• against static analysis (or even dynamic analysis)?
Finite State Machine (simple example)
**Boost Meta State Machine (MSM) library**

```cpp
// --- Transition table
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 , >
//     |       |       |       |       |       |
> {};
```
Result

• **Without obfuscation**

function_to_protect("did", "again");

• **With obfuscation**

OBFUSCATED_CALL(function_to_protect, "did", "again");

• **Even better**

OBFUSCATED_CALL(function_to_protect, OBFUSCATED("did"); OBFUSCATED("again");)
Without obfuscation

```
sub_10000160E proc near
    push    rbp
    mov     rbp, rsp
    call    sub_100001400
    lea     rdi, aDid ; "did"
    lea     rsi, aAgain ; "again"
    pop     rbp
    jmp     sub_1000014F2
sub_10000160E endp
```
With obfuscation

Etc, etc, …
Application 3: FSM + Debugger Detection

- FSM
  - To fight against static analysis
- Debugger detection
  - To fight against dynamic analysis
Finite State Machine

- Follows a different path depending of a predicate (Debugged or not Debugged, that is the question)
More obfuscation

- Obfuscate predicate result
- Avoid simple “if”, too simple for reverse engineers
- Make computation instead
- If the example, counter is odd if predicate is false
More obfuscation

- Obfuscate function address
- Otherwise, IDA is smart enough to get it
- Simply make some computation on address
- Using MetaRandom (like for strings obfuscation)
Predicate

- Debugger detection is only an example
  - In the example, implemented only for Mac OS X / iOS
- Virtual environment detection
- Jailbreak detection
- Etc
Examples

• All code presented here is available on GitHub
  • https://github.com/andrivet/ADVobfuscator

• Contains
  • obfuscator (in source)
  • examples

• BSD 3-clauses license
# Compilers support

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compatible</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple LLVM 5.1</td>
<td>Yes</td>
<td>Previous versions not tested</td>
</tr>
<tr>
<td>Apple LLVM 6.0</td>
<td>Yes</td>
<td>Xcode 6, 6.1 beta</td>
</tr>
<tr>
<td>LLVM 3.4, 3.5</td>
<td>Yes</td>
<td>Previous versions not tested</td>
</tr>
<tr>
<td>GCC 4.8.2 or higher</td>
<td>Yes</td>
<td>Previous versions not tested Compile with -std=c++11</td>
</tr>
<tr>
<td>Intel C++ 2013</td>
<td>Yes</td>
<td>Version 14.0.3 (2013 SP1 Update 3)</td>
</tr>
<tr>
<td>Visual Studio 2013</td>
<td>No</td>
<td>Lack of constexpr support</td>
</tr>
<tr>
<td>Visual Studio 14</td>
<td>Almost</td>
<td>Not far, lack init of arrays CTP3 tested</td>
</tr>
</tbody>
</table>
Compilers options

- Use appropriate compiler options to generate a RELEASE build
  - Xcode: Deployment Postprocessing = Yes
  - GCC: -std=c++11 s -O3

- Otherwise, you will get a binary with original string literals inside

- Disabling RTTI (Runtime Type Information) generates an even more silent binary
  - But not compatible with MSM
My current researches

- More obfuscation areas and techniques
- Apply to Objective-C / Swift
  - selectors
- Apply to Android
  - still using some C++11 and FSM
C++11 metaprogramming applied to software obfuscation
Black Hat Europe 2014 - Amsterdam

Sebastian Andrivet - sebastien@andrivet.com, @AndrivetSeb
CTO, ADVTOOLS SARL, www.advtools.com

Abstract. The C++ language and its siblings like C and Objective-C are one of the most used languages. 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 confused 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 techniques presented rely only on C++11, as standardized by ISO. 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 models. 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 mobile 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 HTML5).

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”. 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.
Contact

- @AndrivetSeb
- sebastien@andrivet.com
Thank you

Questions?