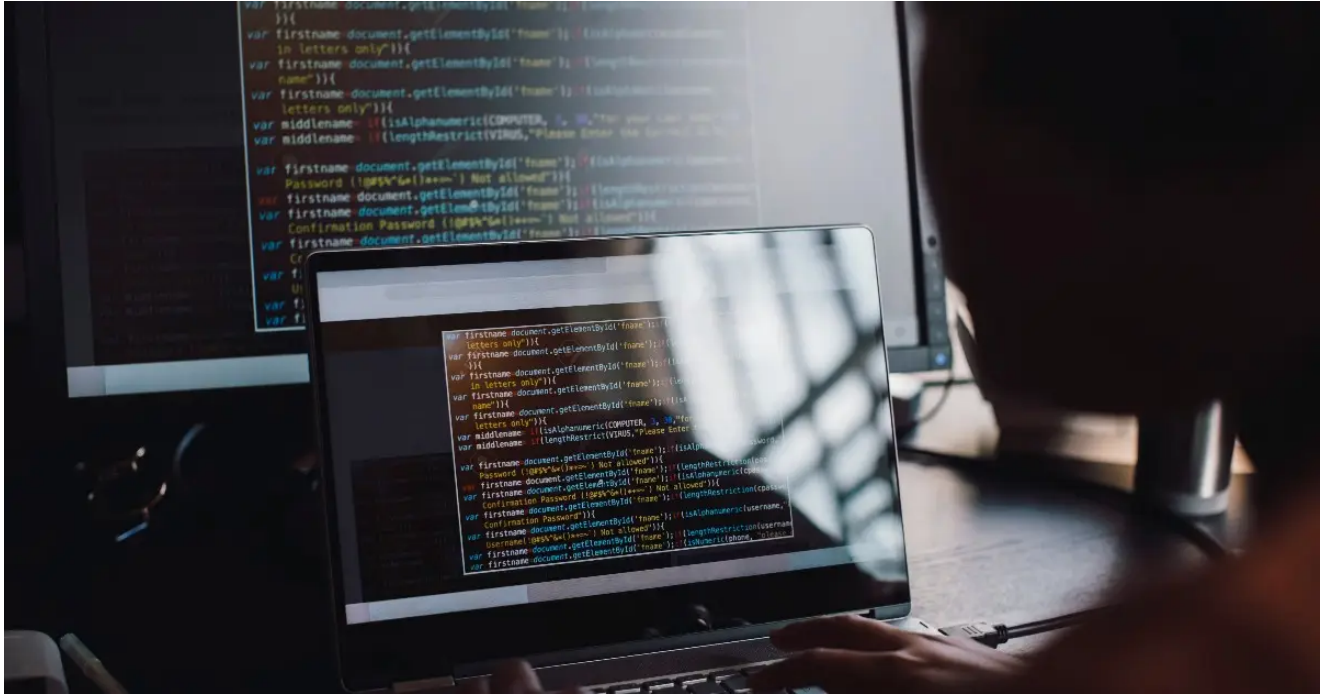


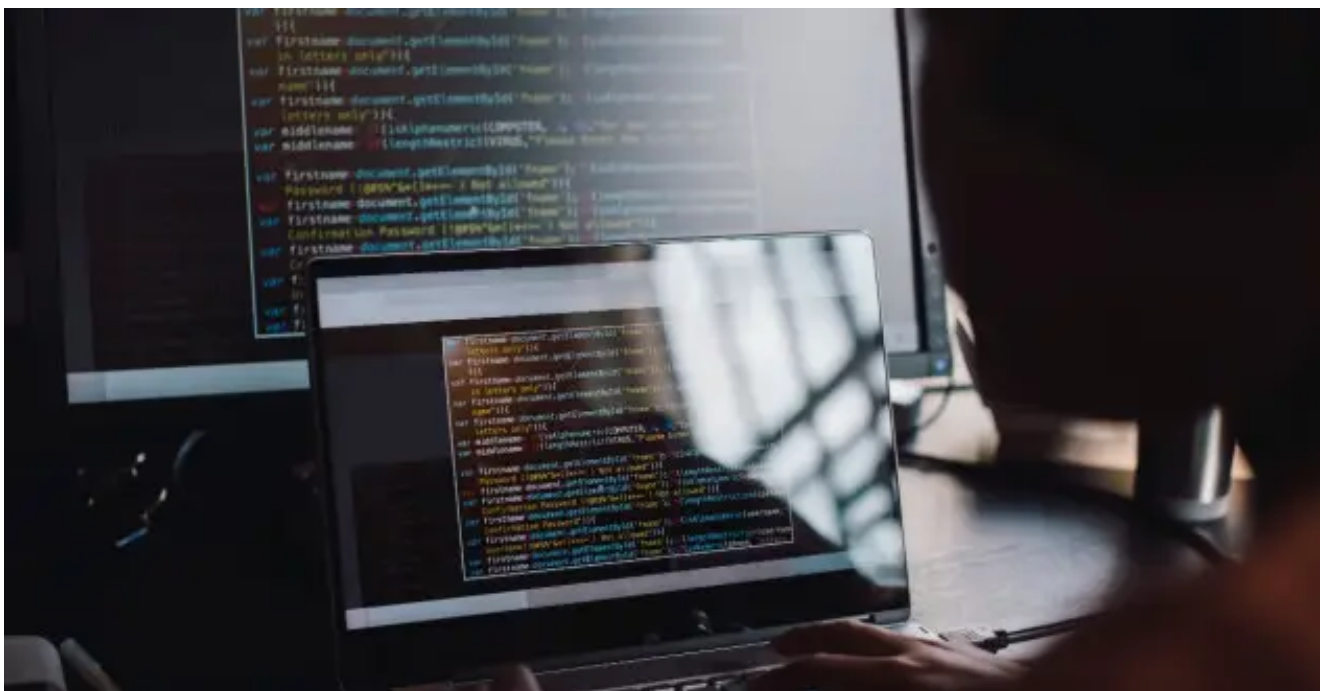
TrickBot Gang Uses Template-Based Metaprogramming in Bazar Malware

 securityintelligence.com/posts/trickbot-gang-template-based-metaprogramming-bazar-malware/



[Home](#) / [Endpoint](#)

TrickBot Gang Uses Template-Based Metaprogramming in Bazar Malware



[Endpoint](#) February 2, 2022

By [Kevin Henson](#) 6 min read

Malware authors use various techniques to obfuscate their code and protect against reverse engineering. Techniques such as control flow obfuscation using [Obfuscator-LLVM](#) and encryption are often observed in malware samples.

This post describes a specific technique that involves what is known as metaprogramming, or more specifically template-based metaprogramming, with a particular focus on its implementation in the Bazar family of malware (BazarBackdoor/BazarLoader). Bazar is best known for its ties to the cybercrime gang that develops and uses [the TrickBot Trojan](#). It is a major cybercrime syndicate that is [highly active](#) in the online crime arena.

A Few Words About Metaprogramming

Metaprogramming is a technique where programs are designed to analyze or generate new code at runtime. Developers typically use metaprogramming techniques to make their code more efficient, modular and maintainable. Template-based metaprogramming incorporates templates that serve as models for code reuse. The templates can be written to handle multiple data types.

For example, the basic function template shown below can be used to define multiple functions that return the maximum of two values such as two numbers or two strings. The type is generalized in the template parameter `<typename T>`, as a result, `a` and `b` will be defined based on the usage of the function. One of the “magical” attributes of templates is that the `max()` function doesn’t actually exist until it’s called and compiled. For the example below, three functions will be created at compile time, one for each call.

```

//Sample function template
template<typename T>
T max (T a, T b)
{
    // if b < a then yield a else yield b
    return b < a ? a : b;
}

// Calls to max()
max(10,5);
max(5.5, 8.9);
max("reverse", "engineering");

```

Templates can be quite complex; however, this high-level understanding will suffice in grasping how the concept is used to a malware author's advantage.

Malware Development

Malware authors take advantage of the metaprogramming technique to both obfuscate important data and ensure that certain elements, such as code patterns and encryption keys, are generated uniquely with each compilation. This hinders analysis and makes developing signatures for static detection more difficult because the encryption code changes with each compiled sample.

The key components in metaprogramming used to accomplish this type of obfuscation are the templates and another feature called constexpr functions. In simple terms, a constexpr function's return value is determined at compile time.

To illustrate how this works, the following sections will compare samples compiled from the open-source library ADVobfuscator to Bazar samples found in the wild. The adoption of more advanced programming techniques within the Bazar malware family is especially relevant since the operators of Bazar are highly active in attacks against organizations across the globe.

ADVobfuscator

To get a better understanding of how template programming is utilized with respect to string obfuscation, let's take a look at two header files from ADVobfuscator. [ADVobfuscator](#) is described as an "Obfuscation library based on C++11/14 and metaprogramming." The MetaRandom.h and MetaString.h header files from the library are discussed below.

MetaRandom.h

The MetaRandom.h header file generates a pseudo-random number at compile time. The file implements the keyword `constexpr` in its template classes. The `constexpr` keyword declares that the value of a function or variable can be evaluated at compile time and, in this example, facilitates the generation of a pseudo-random integer seed based on the compilation time that is then used to generate a key.

```
namespace
{
    // I use current (compile time) as a seed

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

    // Convert time string (hh:mm:ss) into a number
    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;
}
```

Figure 1: Code Block 1 MetaRandom.h

MetaString.h

The MetaString.h header file consists of versions of a template class named *MetaString* that represents an encrypted string. Through template programming, MetaString can encrypt each string with a new algorithm and key during compilation of the code. As a result, a

sample could be produced with the following string obfuscation:

- Each character in the string is XOR encrypted with the same key.
- Each character in the string is XOR encrypted with an incrementing key.
- The key is added to each character of the string. As a result, decryption requires subtracting the key from each character.

Here is a sample MetaString implementation from ADVobfuscator.

This template defines a MetaString with an algorithm number (N), a key value and a list of indexes. The algorithm number controls which of the three obfuscation methods are used and is determined at compile time.

```
template<int N, char Key, typename Indexes>
struct MetaString;
```

Figure 2: Code Block 2 MetaString.h

This is a specific implementation of MetaString based on the above template. The algorithm number (N) is 0, K is the pseudo-random key and I (Indexes) represent the character index in the string. When the algorithm number 0 is generated at compile time, this implementation is used to obfuscate the string. If the algorithm number 1 is generated, the corresponding implementation is used. *ADVobfuscator* uses the C++ macro `__COUNTER__` to generate the algorithm number.

```
template<char K, int... I>
struct MetaString<0, K, Indexes<I...>>
{
    // Constructor. Evaluated at compile time.
    constexpr ALWAYS_INLINE MetaString(const char* str)
        : key_{ K }, buffer_{ encrypt(str[I], K)... } {}

    // Runtime decryption. Most of the time, inlined
    inline const char* decrypt()
    {
        for (size_t i = 0; i < sizeof...(I); ++i)
```

```

        buffer_[i] = decrypt(buffer_[i]);
    buffer_[sizeof...(l)] = 0;
    LOG("— Implementation #" << 0 << " with key 0x" << hex(key_));
    return const_cast<const char*>(buffer_);
}

private:
    // Encrypt / decrypt a character of the original string with the key
    constexpr char key() const { return key_; }
    constexpr char ALWAYS_INLINE encrypt(char c, int k) const { return c ^ k; }
    constexpr char decrypt(char c) const { return encrypt(c, key()); }

    volatile int key_; // key. "volatile" is important to avoid uncontrolled over-optimization by
    the compiler

    volatile char buffer_[sizeof...(l) + 1]; // Buffer to store the encrypted string + terminating
    null byte

};

```

Figure 3: Code Block 3 MetaString.h

ADVobfuscator Samples

Interesting code patterns are observed when samples are built using ADVobfuscator. For example, after compiling the Visual Studio project found in the public [Github repo](#), the resulting code shows the characters of the string being moved to the stack, followed by a decryption loop.

These snippets illustrate the dynamic nature of the library. Each string is obfuscated using one of the three obfuscation methods previously described. Not only are the methods different, the opcodes — the values in blue, which are commonly used in developing YARA rules — can vary as well for the same obfuscation method. This makes developing signatures, parsers and decoders more difficult for analysts. Notably, the same patterns are observed in *BazarLoader* and *BazarBackdoor* samples.

XOR encryption with the same key XOR encryption with an Incrementing key

```

C7 84 24 00 00 00 45 mov [esp+0A0h+key_1], 45h ; '
00 00 00
23 52 xor
C6 84 24 8C 00 00 07 mov [esp+0A0h+encrypted_string], 7
C6 84 24 80 00 00 37 mov [esp+0A0h+var_13], 37h ; '
C6 84 24 8F 00 00 2C mov [esp+0A0h+var_12], 2Ch ; '
C6 84 24 87 00 00 31 mov [esp+0A0h+var_11], 31h ; '
C6 84 24 90 00 00 28 mov [esp+0A0h+var_10], 28h ; '
C6 84 24 91 00 00 20 mov [esp+0A0h+var_9], 20h ; '
C6 84 24 92 00 00 20 mov [esp+0A0h+var_8], 20h ; '
C6 84 24 93 00 00 05 mov [esp+0A0h+var_7], 5h ; '
C6 84 24 94 00 00 16 mov [esp+0A0h+var_6], 16h ; '
C6 84 24 95 00 00 38 mov [esp+0A0h+encrypted_string_11], 38h ; '
C6 84 24 96 00 00 20 mov [esp+0A0h+encrypted_string_10], 20h ; '
C6 84 24 97 00 00 24 mov [esp+0A0h+encrypted_string_10A], 24h ; '
C6 84 24 98 00 00 37 mov [esp+0A0h+encrypted_string_10B], 37h ; '
C6 84 24 99 00 00 36 mov [esp+0A0h+encrypted_string_10C], 36h ; '
8A 84 24 8C 00 00 00 mov al, [esp+0A0h+encrypted_string_10D]
8C 84 24 9A 00 00 00 mov dword ptr [eax+20h]
8E 87 1F 84 00 00 00 nop word ptr [eax+eax*00000000h]
00

loc_401280:
.text:00401280 mov al, [esp+0A0h+encrypted_string]
.text:00401281 mov ecx, [esp+0A0h+key_1]
.text:00401282 xor cl, al
.text:00401283 mov [esp+0A0h+encrypted_string], cl
.text:00401284 inc edi
.text:00401285 cmp edi, 0Eh
.text:00401286 jnb short loc_401280

loc_401116:
.text:00401116 mov cl, [esp+eax+0A0h+encrypted_string_8]
.text:00401117 mov edx, [esp+0A0h+key]
.text:00401118 add d1, al
.text:00401119 xor d1, cl
.text:00401120 mov [esp+eax+0A0h+encrypted_string_8], d1
.text:00401121 inc eax
.text:00401122 cmp eax, 0Eh
.text:00401123 jnb short loc_401116
    
```

Figure 4: Compiled ADVobfuscator Exemplar Samples

BazarBackdoor/BazarLoader

BazarLoader and BazarBackdoor are malware families attributed to the TrickBot threat group, a.k.a. ITG23. Both are written in C++ and compiled for 64bit and 32bit Windows. BazarLoader is known to download and execute BazarBackdoor, and both use the Emercoin DNS domain (.bazar) when communicating with their C2 servers.

Other attributes of the loader and backdoor include extensive use of API function hashing and string obfuscation where each string is encrypted with varying keys. The string obfuscation methodology implemented in these files is interesting when compared with the ADVobfuscator samples previously described.

Bazar String Obfuscation

The string obfuscation implemented in variants of BazarLoader and BazarBackdoor is similar to what is implemented in *ADVobfuscator*. For example, the BazarBackdoor sample *189cbe03c6ce7bdb691f915a0ddd05e11adda0d8d83703c037276726f32dff56* detailed in Figure 5 contains a modified version of the string obfuscation techniques found in *ADVobfuscator*. In Figure 5, the string is moved to the stack four bytes at a time and the key used in the decryption loop is four bytes.

```

.text:004029B0 044 E9 6A ED FF FF call sub_40172C
.text:004029C2 044 64 64 push 64h ; 'd'
.text:004029C4 044 C0 pop ecx
.text:004029C5 044 Encrypted String moved to stack
.text:004029C7 044 call sub_407000
.text:004029CC 044 mov [ebp+var_34], 386BC43h
.text:004029D3 044 B9 00 AD 07 54 mov ecx, 5407AD00h
.text:004029D8 044 C7 45 D0 20 E2 77 31 mov [ebp+var_36], 3177E220h
.text:004029DF 044 88 F0 mov esi, eax
.text:004029E1 044 C7 45 D4 6E FD 75 38 mov [ebp+var_2C], 3875FD6EH
.text:004029E8 044 C7 45 D8 63 C8 74 27 mov [ebp+var_28], 2774C863h
.text:004029EF 044 C7 45 DC 20 CB 66 3D mov [ebp+var_24], 3D66CB20h
.text:004029F6 044 C7 45 E0 6C C8 63 78 mov [ebp+var_20], 7863C86Ch
.text:004029FD 044 C7 45 E4 20 C8 75 26 mov [ebp+var_1C], 2675C820h
.text:00402A04 044 C7 45 E8 6F DF 27 37 mov [ebp+var_18], 3727DF6FH
.text:00402A0B 044 C7 45 EC 6F C9 62 6E mov [ebp+var_14], 6E62C96FH
.text:00402A12 044 C7 45 F0 20 88 63 5E mov [ebp+var_10], 5E638820h
.text:00402A19 044 89 40 F4 mov [ebp+var_C], ecx
.text:00402A1C 044 88 45 C0 mov eax, [ebp+var_34]
.text:00402A1F 044 88 5D F8 mov [ebp+var_8], bl
.text:00402A22 044 38 5D F8 cmp [ebp+var_8], bl
.text:00402A25 044 75 10 jnz short loc_402A37

```

```

.text:00402A27 Decryption Loop loc_402A27:
.text:00402A27 044 8B 44 90 C0 mov eax, [ebp+ebx*4+var_34]
.text:00402A2B 044 33 C1 xor eax, ecx
.text:00402A2D 044 89 44 90 C0 mov [ebp+ebx*4+var_34], eax
.text:00402A31 044 43 inc ebx
.text:00402A32 044 83 FB 0B cmp ebx, 0Bh
.text:00402A35 044 72 F0 jb short loc_402A27

```

Figure 5: XOR String Decryption 1

```

.text:00408B40 738 C7 85 58 FA FF FF D8 B0 5B 12 mov [ebp+var_5A5], 1258B000h
.text:00408B55 738 C7 85 5C FA FF FF CA AB 4B 2B mov [ebp+var_5A4], 2B4BABCah
.text:00408B6F 738 C7 85 60 FA FF FF C5 BE 18 6F mov [ebp+var_5A0], 6F18BEC5h
.text:00408B89 738 C7 85 64 FA FF FF AB 51 62 mov [ebp+var_59C], 6251ABF5h
.text:00408BD3 738 C7 85 68 FA FF FF BF BA 5D 30 mov [ebp+var_598], 305DBA8Fh
.text:00408BD8 738 C7 85 6C FA FF FF D0 BC 4A 31 mov [ebp+var_594], 314ACDD0h
.text:00408BE7 738 C7 85 70 FA FF FF F0 FD 51 1F mov [ebp+var_590], 1F51FDF0h
.text:00408BF1 738 C7 85 74 FA FF FF 90 F9 4F 30 mov [ebp+var_58C], 304FF990h
.text:00408BF8 738 C7 85 78 FA FF FF C2 AD 5D 6F mov [ebp+var_588], 6F5DADC2h
.text:00408C05 738 C7 85 7C FA FF FF C3 B6 4B 36 mov [ebp+var_584], 364BB6C3h
.text:00408C0F 738 C7 85 80 FA FF FF 88 FD 51 32 mov [ebp+var_580], 3251FD88h
.text:00408C19 738 C7 85 84 FA FF FF 85 BA 57 2C mov [ebp+var_57C], 2C57BA85h
.text:00408C23 738 C7 85 88 FA FF FF DP BC 56 36 mov [ebp+var_578], 3656BCDFh
.text:00408C2B 738 C7 85 8C FA FF FF 90 F8 38 42 mov [ebp+var_574], 4238F890h
.text:00408C37 738 B8 85 C0 F9 FF FF mov eax, [ebp+var_640]
.text:00408C3D 738 C6 85 90 FA FF FF 00 mov [ebp+var_570], 0
.text:00408C44 738 B8 80 90 FA FF FF 00 cmp [ebp+var_570], 0
.text:00408C48 738 75 10 jnz short loc_408C68

```

```

loc_408C4F:
.text:00408C4F 738 8B 84 8D C0 F9 FF FF mov eax, [ebp+ecx*4+var_640]
.text:00408C56 738 35 AB D9 38 42 xor eax, 4238D9ABh
.text:00408C5B 738 89 84 8D C0 F9 FF FF mov [ebp+ecx*4+var_640], eax
.text:00408C62 738 41 inc ecx
.text:00408C63 738 83 F9 34 cmp ecx, 34h ; '4'
.text:00408C66 738 72 E7 jb short loc_408C4F

```

Figure 6: XOR String Decryption 2

TrickBot and Bazar — Ongoing Code Evolution

Based on the similarities discovered through the analysis performed by X-Force, it is evident that the authors of BazarLoader and BazarBackdoor malware utilize template-based metaprogramming. While it is possible to break the resulting string obfuscation, the ultimate intent of the malware author is to hinder reverse engineering and evade signature-based detection. Metaprogramming is just one tool in the threat actors' toolbox. Understanding how these techniques work helps reverse engineers create tools to increase the efficiency of analysis and stay in step with the constant threat malware poses.

Kevin Henson

Malware Reverse Engineer, IBM

Kevin joined IBM Security's X-Force IRIS team as a Malware Reverse Engineer in November 2018 after 21 years of experience in supporting various commercial,...

think 2022



IBM Think Broadcast
Let's think together.

Watch on demand →

