Malware development part 6 - advanced obfuscation with LLVM and template metaprogramming

Introduction

This is the sixth post of a series which regards the development of malicious software. In this series we will explore and try to implement multiple techniques used by malicious applications to execute code, hide from defenses and persist.
Today we will explore anti-disassembly obfuscation using LLVM and template metaprogramming.

LLVM obfuscation

LLVM is a compiler infrastructure. To understand what it is exactly we need to dive into compilation process (this is most accurate for unmanaged code like C/C++).

We can distinguish three steps of assembly generation from the source code:

1. Front end, which includes:
    - scanner, which performs lexical analysis of the code and produces tokens (strings with certain meaning)
    - parser, which produces an abstract syntax tree (tokens grouped in a tree which represents the actual algorithm implemented in the source code)
    - semantic analysis (mainly type checking), during which the AST is checked for errors like wrong use of types or use of variables before initialization
    - generation of intermediate representation, usually based on AST
2. Optimization, which aims at reducing code complexity for example by precalculating stuff. Optimization must not change the algorithm/program itself.
3. Back end, which translates the intermediate representation to expected output (assembly or bytecode).

The core of LLVM is the optimizer but the project also includes a compiler front end - `clang` - which is intended to be used with the LLVM toolchain.

Obfuscator-LLVM

We will leverage `Obfuscator-LLVM` project which is an open-source fork of the LLVM.

Obfuscation works on the mentioned intermediate representation (IR) level. In other words it's a kind of 'anti'-optimization. `Clang` is used to generate IR from source code, then the IR is processed to obfuscate code flow and finally the assembly is generated.

Setup

Having gone through the theoretical introduction, let's prepare the environment for C++ code obfuscation. The Obfuscator-LLVM needs to be downloaded and compiled. The latest branch is `llvm-4.0` (from 2017, the latest version of LLVM is `11.0` nowadays) and the code needs to be compiled with Visual Studio 2017 and not 2019 (as it gives some errors during compilation). We need to use `CMake` to generate VS2017 project and then compile it (minding the target architecture). We can use `Developer Command Prompt for VS 2017` which is a part of Visual Studio 2017:

```
git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator
cd obfuscator
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
```

Note: I had to manually define `ENDIAN_LITTLE` identifier to get rid of some compilation errors.

There are different ways to use `Obfuscator-LLVM` compiler:

- use manually via command line
- add the compiler as a *custom build tool* for `.cpp` and other files in Visual Studio (in a relevant file *Property Pages*)
- use VS Installer to install a `clang-cl` platform toolset and manually swap Visual Studio's clang version with the compiled compiler (this kinda sounds like a chicken-egg problem :))

Usage and features

Let's write a simple program which performs some rather simple calculations based on a pseudorandom value:

```
int main()
{
	int a = GetTickCount64();
	int b = a % 10;
	int c = 0;
	for (int i = 0; i < b; i++)
	{
		c += a % i;
	}
	return c;
}
```

Note: I compiled this code without `CRT` dependency so the binary is small and there's no additional code (like `mainCRTStartup` etc.) - see part 4 of malware development series.

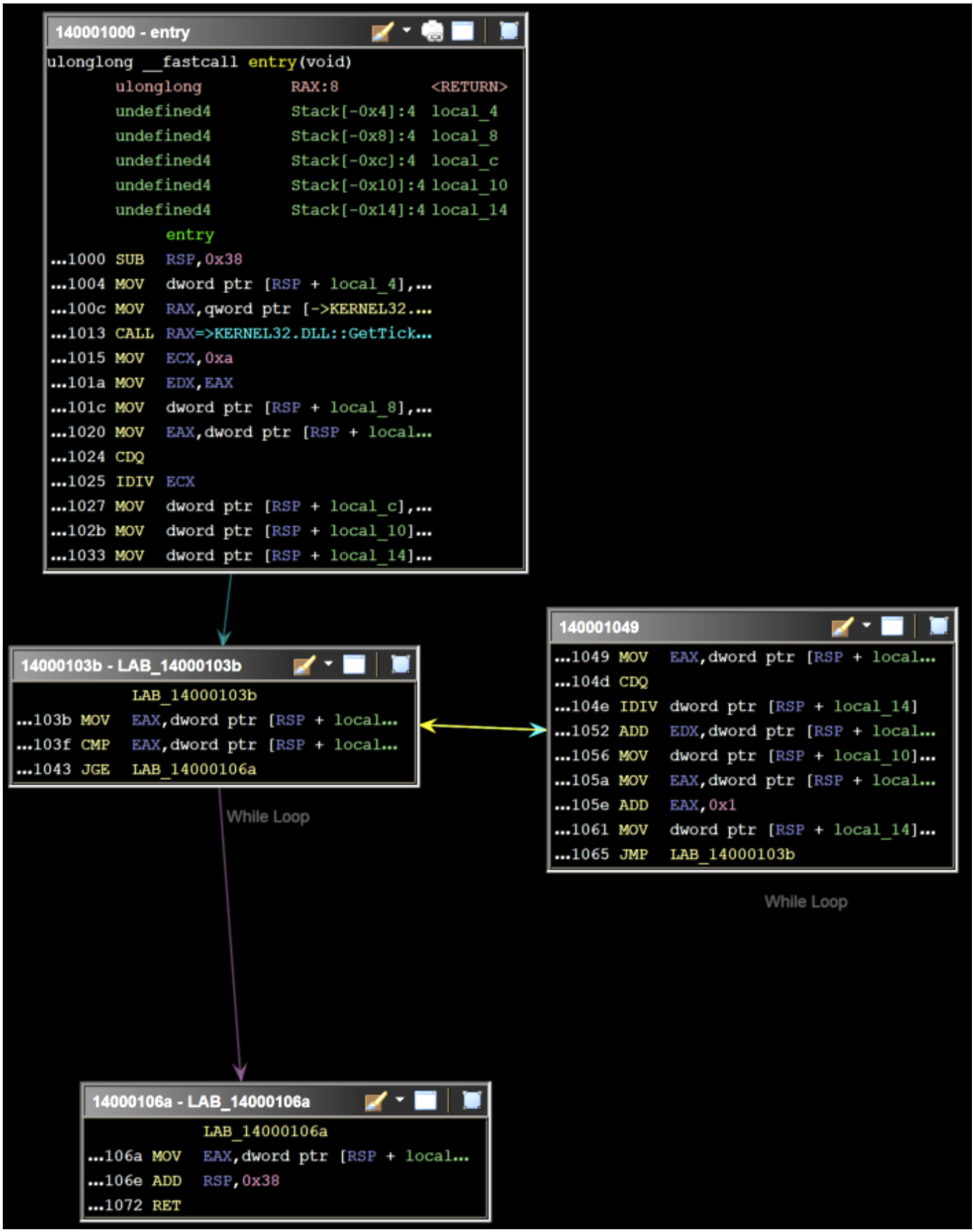
This is how the code looks like after decompiling with Ghidra:

```
ulonglong entry(void)

{
  ULONGLONG UVar1;
  int local_14;
  uint local_10;

  UVar1 = GetTickCount64();
  local_10 = 0;
  local_14 = 0;
  while (local_14 < (int)UVar1 % 10) {
    local_10 = (int)UVar1 % local_14 + local_10;
    local_14 = local_14 + 1;
  }
  return (ulonglong)local_10;
}
```

And the program graph:



`Obfuscator-LLVM` has 3 code obfuscation features: instructions substitution, bogus control flow and control flow flattening. Let's explore them. Details can be found in the project's repository

These features use random value which has to be provided as a command line parameter (`-mllvm -aesSeed=1234567890ABCDEF1234567890ABCDEF`) on Windows systems (on Linux it uses `/dev/random`).

Instructions substitution

This replaces simple arithmetic operations with more complex but equivalent ones. For example: `a = b + c` may be changed to `r = rand(); a = b + r; a = a + c; a = a - r;`. The random value is calculated during the compilation.

It's possible to apply substitutions multiple times. Random seed from the command line is used to randomly select substitute instruction sequence so this brings some additional uniqueness to the resulting binary.

Let's add following switches to the compilation command line: `-mllvm -sub -mllvm -sub_loop=5 -mllvm -aesSeed=1234567890ABCDEF1234567890ABCDEF`

Resulting assembly (decompiled):

```
ulonglong entry(void)

{
  ULONGLONG UVar1;
  int local_14;
  uint local_10;

  UVar1 = GetTickCount64();
  local_10 = 0;
  local_14 = 0;
  while (local_14 < (int)UVar1 % 10) {
    local_10 = -((-(local_10 + 0x1ff53be2) - (int)UVar1 % local_14) + 0x1ff53be2);
    local_14 = -0x2eece4c7 - (-1 - (0x662d2b91 - (0x374046ca - local_14)));
  }
  return (ulonglong)local_10;
```

And the graph:

bfuscator ‘deoptimizations’ quite well.

This adds opaque predicates before instruction blocks. An opaque predicate is basically a portion of (prefably random) code which is evaluated at the runtime to a predetermined logical value (`true` or `false`). It is followed by a conditional jump which points to an original instruction block.

This obfuscation can also be applied multiple times, and can target random blocks of code.

Example usage: `-mllvm -bcf -mllvm -bcf_prob=100 -mllvm -bcf_loop=1 -mllvm -aesSeed=1234567890ABCDEF1234567890ABCDEF`

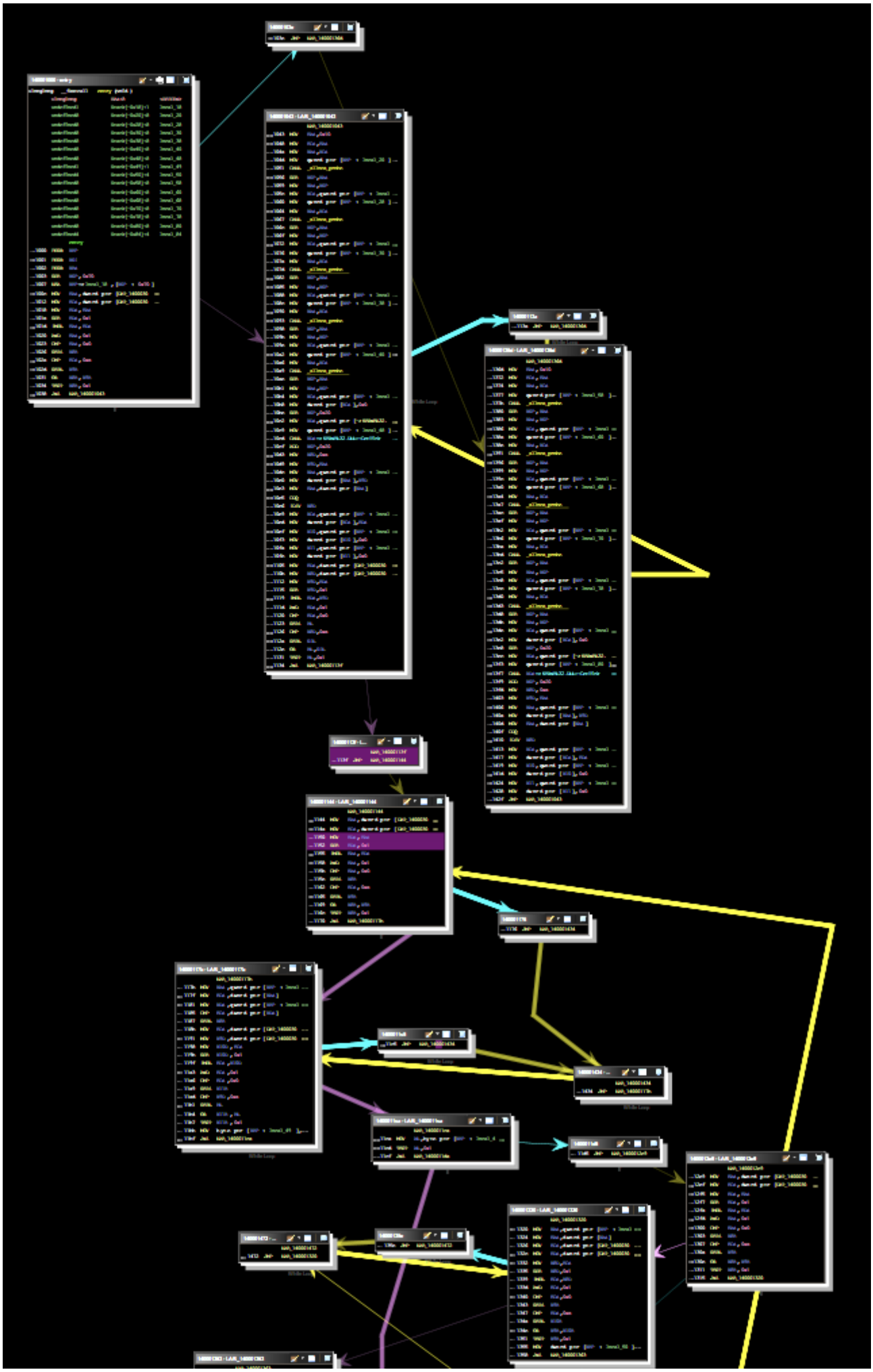
Resulting assembly (decompiled):

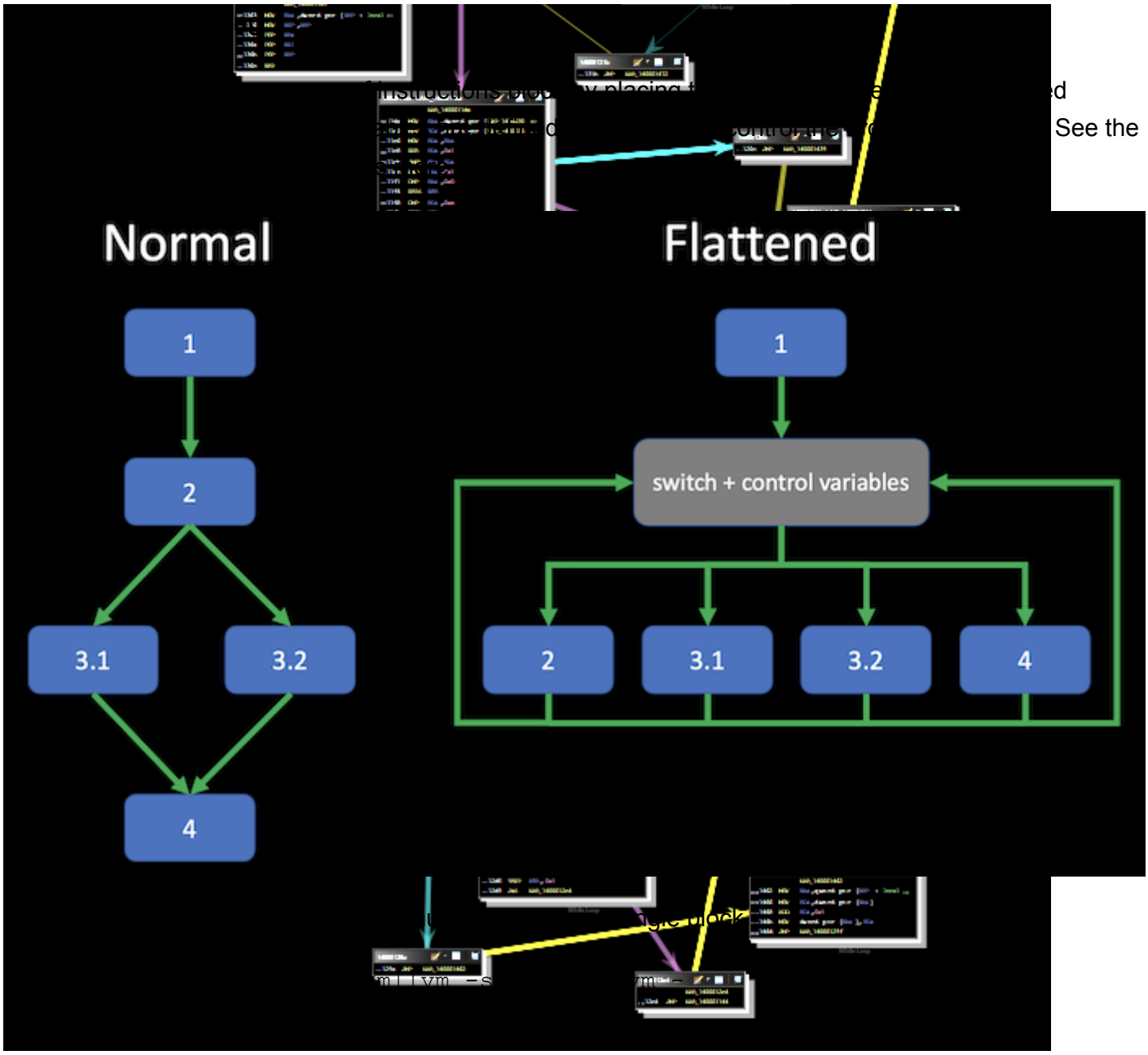
```
ulonglong entry(void)
{
  int *piVar1;
  ULONGLONG UVar2;
  undefined *puVar3;
  undefined *puVar4;
  undefined auStack136 [8];
  undefined4 *local_80;
  undefined4 *local_78;
  int *local_70;
  int *local_68;
  undefined *local_60;
  undefined8 local_58;
  int *local_48;
  uint *local_40;
  int *local_38;
  int *local_30;
  undefined *local_28;
  undefined8 local_20;

  puVar3 = auStack136;
  puVar4 = auStack136;
  if ((DAT_140003010 * (DAT_140003010 + -1) & 1U) == 0 || DAT_140003014 < 10) goto LAB_1400
  do {
    local_58 = 0x10;
    *(undefined8 *)(puVar4 + -8) = 0x140001380;
    local_60 = puVar4 + -0x20;
    *(undefined8 *)(puVar4 + -0x28) = 0x140001396;
    local_68 = (int *)(puVar4 + -0x40);
    *(undefined8 *)(puVar4 + -0x48) = 0x1400013ac;
    local_70 = (int *)(puVar4 + -0x60);
    *(undefined8 *)(puVar4 + -0x68) = 0x1400013c2;
    local_78 = (undefined4 *)(puVar4 + -0x80);
    *(undefined8 *)(puVar4 + -0x88) = 0x1400013d8;
    puVar3 = puVar4 + -0xa0;
    *(undefined4 *)(puVar4 + -0x20) = 0;
    local_80 = (undefined4 *)(puVar4 + -0xa0);
    *(undefined8 *)(puVar4 + -200) = 0x1400013f9;
    UVar2 = GetTickCount64(puVar4[-200]);
    piVar1 = local_68;
    *local_68 = (int)UVar2;
    *local_70 = *piVar1 % 10;
    *local_78 = 0;
    *local_80 = 0;
```

```
LAB_140001043:
    local_20 = 0x10;
    *(undefined8 *)(puVar3 + -8) = 0x140001056;
    local_28 = puVar3 + -0x20;
    *(undefined8 *)(puVar3 + -0x28) = 0x14000106c;
    local_30 = (int *)(puVar3 + -0x40);
    *(undefined8 *)(puVar3 + -0x48) = 0x140001082;
    local_38 = (int *)(puVar3 + -0x60);
    *(undefined8 *)(puVar3 + -0x68) = 0x140001098;
    local_40 = (uint *)(puVar3 + -0x80);
    *(undefined8 *)(puVar3 + -0x88) = 0x1400010ae;
    puVar4 = puVar3 + -0xa0;
    *(undefined4 *)(puVar3 + -0x20) = 0;
    local_48 = (int *)(puVar3 + -0xa0);
    *(undefined8 *)(puVar3 + -200) = 0x1400010cf;
    UVar2 = GetTickCount64(puVar3[-200]);
    piVar1 = local_30;
    *local_30 = (int)UVar2;
    *local_38 = *piVar1 % 10;
    *local_40 = 0;
    *local_48 = 0;
  } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
  while( true ) {
    do {
    } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
    if (*local_38 <= *local_48) break;
    if ((DAT_140003010 * (DAT_140003010 + -1) & 1U) == 0 || DAT_140003014 < 10) goto LAB_140001211;
    do {
      *local_40 = *local_40 + *local_30 % *local_48;
LAB_140001211:
      *local_40 = *local_30 % *local_48 + *local_40;
    } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
    if ((DAT_140003010 * (DAT_140003010 + -1) & 1U) == 0 || DAT_140003014 < 10) goto LAB_14000129f;
    do {
      *local_48 = *local_48 + 1;
LAB_14000129f:
      *local_48 = *local_48 + 1;
    } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
  }
  do {
  } while ((DAT_140003010 * (DAT_140003010 + -1) & 1U) != 0 && 9 < DAT_140003014);
  return (ulonglong)*local_40;
}
```

And the graph:



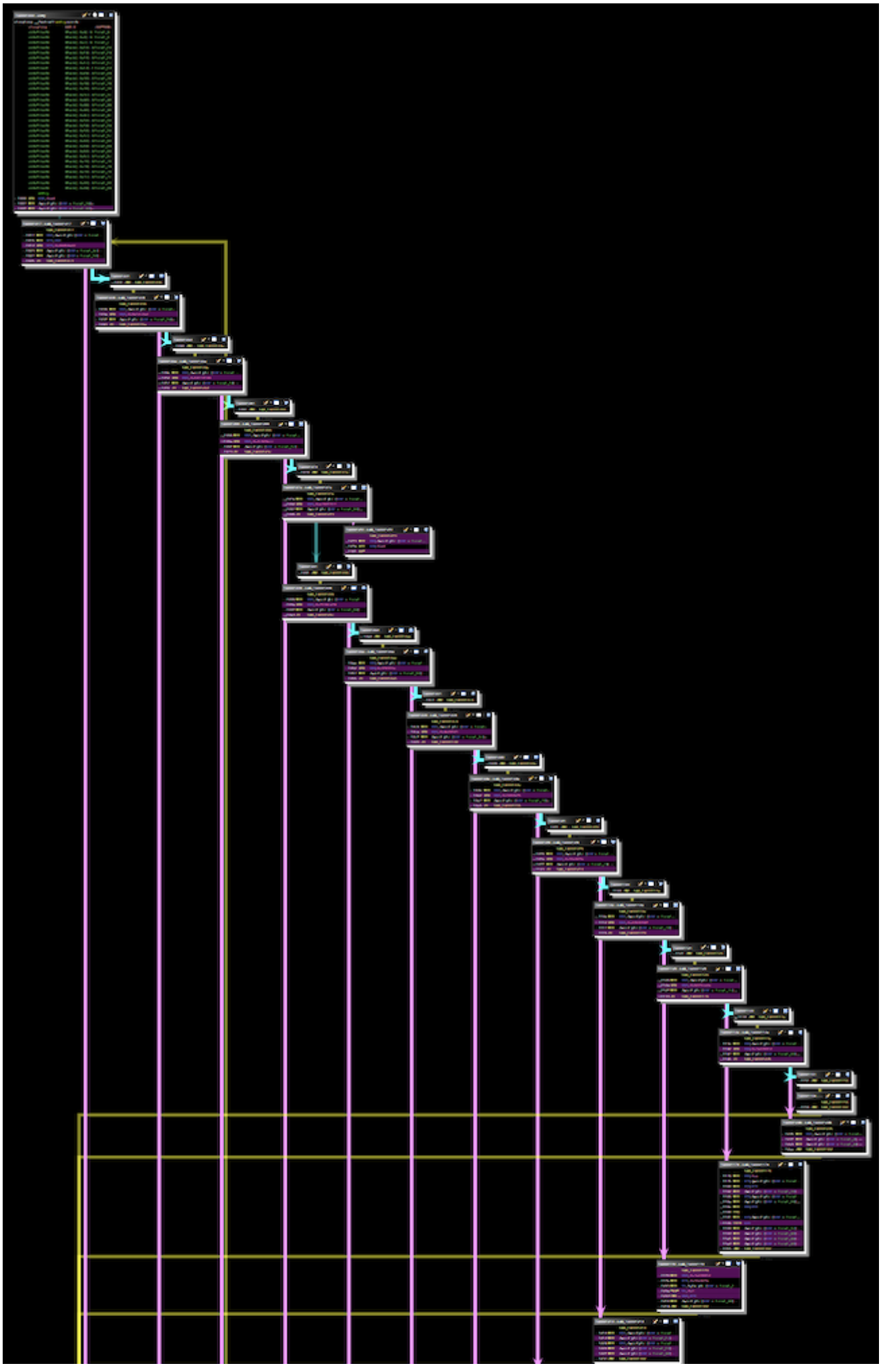


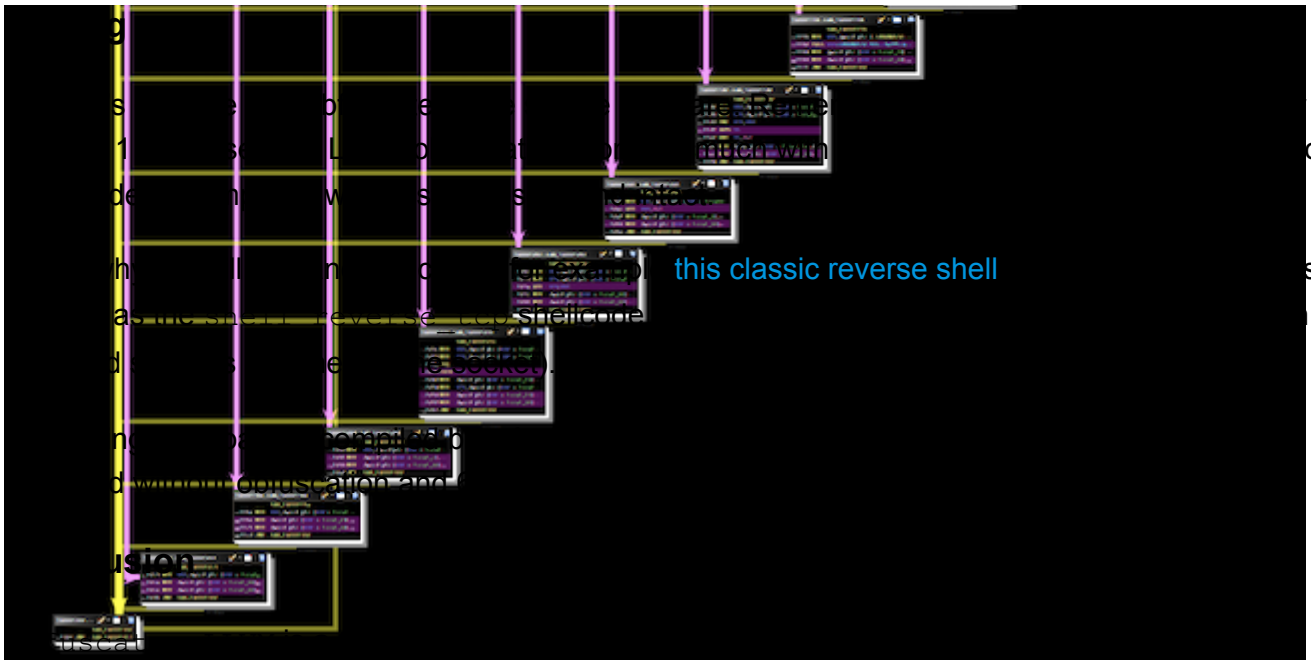
Resulting assembly (decompiled):

```
ulonglong entry(void)
{
  int local_48;
  int local_44;
  int local_40;
  int local_3c;
  int local_38;
  ULONGLONG local_30;
  int local_24;
  bool local_1d;
  int local_1c;
  int local_18;
  int local_14;
  int local_10;
  int local_c;
  int local_8;
  uint local_4;

  local_48 = 0x589dafb;
  while( true ) {
    while( true ) {
      while( true ) {
        while( true ) {
          while (local_48 == -0x797495de) {
            local_44 = local_8;
            local_48 = 0x9a12c2a2;
          }
          if (local_48 != -0x65ed3d5e) breal
          local_24 = local_44;
          local_48 = 0x4e28521;
        }
        if (local_48 != -0x49eefe7a) break;
        local_c = local_44;
        local_48 = 0x3f8355e;
      }
      if (local_48 != -0x236d6134) break;
      local_14 = local_1c % local_18;
      local_10 = local_40;
      local_48 = 0xf534caf4;
    }
    if (local_48 == -0x1872eae9) break;
    if (local_48 == -0xacb350c) {
      local_40 = local_10 + local_14;
      local_48 = 0xb6110186;
    }
    else {
      if (local_48 == 0x3f8355e) {
        local_8 = local_c + 1;
        local_48 = 0x868b6a22;
      }
      else {
        if (local_48 == 0x4e28521) {
          local_1d = local_24 < local_3c;
          local_48 = 0x2d6d294f;
        }
        else {
          if (local_48 == 0x589dafb) {
            local_30 = GetTickCount64();
            local_48 = 0x62d3ce2a;
          }
          else {
            if (local_48 == 0x5be82fa) {
              local_1c = local_38;
              local_18 = local_44;
              local_48 = 0xdc929ecc;
            }
            else {
              if (local_48 == 0x2d6d294f) {
                local_48 = 0x7a2d8912;
                if ((local_1d & 1U) != 0) {
                  local_48 = 0x5be82fa;
                }
              }
              else {
                if (local_48 == 0x62d3ce2a) {
                  local_38 = (int)local_30;
                  local_3c = local_38 % 10;
                  local_40 = 0;
                  local_44 = 0;
                  local_48 = 0x9a12c2a2;
                }
                else {
                  if (local_48 == 0x7a2d8912) {
                    local_4 = local_40;
                    local_48 = 0xe78d1517;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
  return (ulonglong)local_4;
}
```

And the graph:





from ors this classic reverse shell same its code

compilation and how can this process be modified to make static assembly analysis more difficult and time-consuming. However it's important to remember that the IR-level obfuscation can be reversed (not completely, but still). See this great article for an example of the deobfuscation process.

Here are some general thoughts and considerations: From an offensive penetration tester perspective, it's important to combine multiple layers of code protection measures to minimize chances of detection and hinder manual analysis as much as possible (well, with a reasonable ammount of our efforts). This helps to deliver effective adversary emulations focused on the actual objectives. Of course more advanced malware requires more work put into it by defensive teams, which is also a good thing.

Anyway, make sure to consider implementing some intermediate representation level obfuscation into your offensive tooling build process.

Other LLVM-based obfuscators

Also be sure to check other LLVM-based obfuscators and articles on building custom obfuscators with LLVM:

https://github.com/HikariObfuscator/Hikari/

https://medium.com/@polarply/build-your-first-llvm-obfuscator-80d16583392b

http://www.babush.me/dumbo-llvm-based-dumb-obfuscator.html

https://github.com/emc2314/YANSOllvm

https://blog.scrt.ch/2020/06/19/engineering-antivirus-evasion/

https://blog.scrt.ch/2020/07/15/engineering-antivirus-evasion-part-ii/

Template metaprogramming

Before diving into the details of C++ constructs like templates, constant expressions and metaprogramming, let's consider a simple case: we have a source code with some string literals (like IP addresses, domain names etc.) that need to be obfuscated so they are invisible in the assembly and only revealed at runtime. Easiest thing to do here is to encrypt these literals and replace them with a call to decryption routine, for example:

```
const char* address = "www.example.com";
```

replaced with:

```
char* Decrypt(const char* data);
(...)
char* addr = Decrypt("xxx.yyyyyyy.zzz");
```

Of course we would have to consider string length, null-byte terminators etc.

We would prefer to use plaintext values in the source code and obfuscate/encrypt them automatically during the build process. Replacement of plain strings with encrypted ones can be automated with a pre-build task, e.g. some Python script. But there's another, cooler way to do this.

Introduction

Let's get familiar with some features introduced in `C++11` standard: `template`s and `constexpr`essions. The following won't cover all the details of metaprogramming concepts - it's just a simple introduction which will help to understand how obfuscation based on template metaprogramming actually work.

Templates

Templates are functions that operate on generic types. Templates allow simple creation of functions which operate on multiple types (basic types, structs, classes). For example we can use the following template:

```
template <typename T>
bool Equal(T arg1, T arg2)
{
        return (arg1 == arg2)
}
```

instead of defining overloaded functions:

```
bool Equal(int arg1, int arg2);
bool Equal(double arg1, double arg2);
```

And example template usage:

```
Equal <int>(1, 2));
```

Of course types must implement == operator in order to use the `Equal` function template.

Templates can be also used to create a generic struct or class, which then can be instantiated to be used with a specific type:

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

Stack<Fruit> fruitStack;
Stack<Vegetable> vegetableStack;
```

This also provides type safety, in this case you won't be able to mix fruits with vegetables - `fruitStack.push(new Vegetable());` will produce a compilation error.

Let's see an another example - usage of template for recursive factorial calculation:

```
template <int N>
struct Factorial
{
        enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0>
{
        enum { value = 1 };
};

Factorial<5>::value // 5! = 120
```

We see here that an integer can be a template argument and that a template **specialization** (`template <>`) is needed to define a value for a specific argument.

Constant expressions

The `constexpr` specifier indicates that the value of some expression can be evaluated at compile time. For example, when such a constant expression defined:

```
constexpr int sum(int a, int b)
{
```

```
        return (a + b);
}
```

`Sum(1+2)` will be precalculated at compile time - this calculation won't consume resources at the application's runtime.

Metaprogramming

Metaprogramming is just modifying programs by other programs or by themselves. Turns out that templates are a kind of functional programming language and can be used by compiler to generate source code.

Remember? It's exactly what we were doing with pre-build scripts - creating a temporary source code with sensitive data obfuscated.

String obfuscation

Having understood the ability to write code which can be executed by compilers, let's create a simple string obfuscator which will replace plaintext data with XORed values just before compilation. We would like to use the obfuscation in the following manner: `Obfuscated("secret");`. The `Obfuscated` macro should replace the `"secret"` with a decryption function with an encrypted argument: `Decrypt_runtime(Encrypt_compiletime(secret))`.

To use constant string at compile time, we need to know its exact length. So we will need a compile time function which operates on this length value. So first, we need to create a template which will get an integer as an argument: `template <unsigned int N>`.

Now we will create a struct which holds the obfuscated string (which will replace the plaintext in the source code) and has a compile time function (`constexpr`) as a constructor to obfuscate the plaintext:

```
struct Obfuscator
{
        char data[N] = { 0 };
        constexpr Obfuscator(const char* plaintext)
        {
                for (int i = 0; i < N; i++)
                {
                        data[i] = plaintext[i] ^ 0x00;
                }
        }
}
```

Now we obfuscate data in source code by creating an `Obfuscator<7>` struct from the `Obfuscator<N>` template (7 = string length + null byte):

```
constexpr Obfuscator<7> obfuscated = Obfuscator<7>("secret");
```

To actually use the data in the application we need to decrypt it, so we add deobfuscation function (which operates on a constant value, hence the `const` identifier following its declaration) to the `Obfuscator` template:

```
const char* Deobfuscate() const
{
        char plaintext[N] = { 0 };
        for (int i = 0; i < N; i++)
        {
                plaintext[i] = data[i] ^ 0x11;
        }
        return plaintext;
}
```

Now we can deobfuscate the `obfuscated` constant variable: `obfuscated.Deobfuscate()`.

The last thing to do is to create a helper macro which simplifies the obfuscation in the source code. We will use another goodie of `C++11` - lambda functions:

```
#define Obfuscated(string) []() -> const char* \
{ \
        constexpr auto secret = Obfuscator<sizeof(string) /
sizeof(string[0])>(string); \
        return secret.Deobfuscate(); \
}()
```

Thanks to this string literals appearing in the binary are XOR encrypted. It's possible to enhance this method to make the application create stack based strings which won't appear in the `.text` section of PE file.

Other possibilities

It's possible to implement quite advanced string and code obfuscation using template metaprogramming. For more detailed explanation see this awesome workpaper by Sebastien Andrivet and his ADVobfuscator tool which implements described concepts. There is a number of such obfuscators available and the best thing about them is that we can use them by just adding header files to the project:

https://github.com/fritzone/obfy

https://github.com/revsic/cpp-obfuscator

Summary

This post was just an introduction to advanced and powerful obfuscation methods which leverage LLVM compiler infrastructure and template metaprogramming.

Next time we will talk about keyloggers and implement one.

Written on January 25, 2021