+++ title = "Red Teamer's Guide to Malware Development 5: Malware Development Toolchains" date = 2024-05-26T15:44:35-05:00 draft = false +++

> This is a guide for malware development for ethical red teaming. Nowadays, there are lots of paid courses that cover these same concepts, but fortunately for everyone, these are not necessary to get up to speed. Most of the techniques these courses cover are open source.

The following topics will be covered in this entry:

- Toolchains: Compilers, linkers, assemblers, and build systems.
- Obfuscating compilers.
- Case studies of malware build toolchains in the wild.

C/C++ Compilation Toolchains

In development, the word *toolchain* refers to the set of components that are used to translate higher level code into a finished product that can run on a system. These toolchains can be very simple or exceedingly complex.

At a high level, you have four important components in a typical C/C++ development toolchain:

- Management/Build Systems: Tools like CMake, Visual Studio / MSBuild, Make, Ninja, etc.
- Compilers: Translate high level code into assembly.
- Assemblers: Translate the assembly language into machine code.
- Linkers: Combine outputs from compilers/assemblers into final packaged file(s).

Management & Build Systems

Management/Build tools manage the compilation process. These exist to provide a streamlined way to perform the actual compiling, assembling, and linking, as the commands to these tools become very complex very quickly. This issue compounds massively when different platforms and release types are included in a project.

The most common toolchains we will encounter for maldev purposes are CMake, MSBuild, and Make.

- Make: Old-school build tool. Text files are used to instruct other tools in the toolchain what to do in an abstract way.
- CMake: Advanced version of Make with many more features.
- MSBuild: Microsoft build tool, integrated with Visual Studio. XML files are used to describe the building process in an abstract way.

Compilers

The main compilers we will work with in Windows C/C++ maldev are the following:

- `cl.exe`: Microsoft compiler for C/C++. Closed source.
- `mingw`: GCC with Windows compatibility. Open source.
- Clang/`llvm` (Low-Level Virtual Machine): Highly advanced compiler with other toolchain features. Generates an "intermediate representation" during compilation process. Allows advanced symbolic manipulation. Clang is the compiler frontend commonly used for LLVM.

Assemblers

Assemblers translate assembly code into machine code. E.g., the following code is a human-readable representation of machine code (assembly) for the `uasm64` assembler:

```
.x64
option win64:0x08, casemap:none, frame:auto, stackbase:rsp
TEXT$00 SEGMENT ALIGN(10h) 'CODE' READ WRITE EXECUTE

Main PROC
    xor eax, eax
    ret
Main ENDP

TEXT$00 ENDS

END
```

The instructions inside the `Main PROC` section are the actual code instructions. The assembler will translate these into the following bytes:

```
0x48, 0x31, 0xC0,         // xor rax, rax
0xC3                      // ret
```

The other components instruct the assembler on the file format to generate the code within.

Common Assemblers

There are a handful of assemblers used commonly on the Windows platform:

- MASM and MASM64: Microsoft assemblers. Closed source. MASM64 has a limited feature set compared to the original MASM.
- NASM: Netwide assembler. Open source. Limited feature set.
- FASM: Flat assembler. Minimal open source assembler.
- GAS: GCC's assembler (cross platform). Open Source.
- UASM: Continuation of MASM style assembler with advanced features.

My preferred assembler is UASM64. It uses the MASM syntax and has support for high level abstraction.

Linkers

Linkers combine the output from compilers and assemblers into final executables. Linkers perform manipulation of symbols in these output files to merge them and make a functional final project.

Common Linkers

- Microsoft linker `link.exe`: Closed-source Microsoft linker. Tight integration with Visual Studio.

- GNU `ld`: GCC's linker, available to Windows through MinGW. Open source.
- LLVM Linker `lld`: LLVM's linker. Open source and highly flexible.

Malware Development Toolchains

Malware development poses interesting constraints on how we translate our source code into a final project. This is because of the need for products that can evade defense tools and fulfill unusual needs faced when executing malicious code. Some examples of unusual requirements in maldev are the following:

- Position Independent Shellcode: Machine code that has zero dependencies and can run in a self-sufficient manner. Ordinary Windows code is highly dependent on features provided by the operating system and runtime libraries.
- Obfuscation & Evasion: To evade defense tools, obfuscation of final products is often needed. This means obscuring the intent and form of the malware in order to evade defenses. This can take the format of obfuscated code with dead-end and junk instructions, or the hiding of the actual malicious code within other code.

Case Study: Toolchain for Conti Ransomware

The leaked source code project of the Conti ransomware can serve as a prime example of a complete malware project utilizing the Visual Studio toolchain.

Please use caution with these files and only open on a clean virtual machine. You can find the leak here: Conti Leak.

The folder structure looks like this:

```
C:.
├───cryptor
│   ├───antihooks
│   ├───api
│   ├───chacha20
│   ├───Debug
│   ├───filesystem
│   ├───global
│   ├───logs
│   ├───network_scanner
│   ├───obfuscation
│   ├───prockiller
│   ├───Release
│   ├───threadpool
│   └───x64
│       ├───Debug
│       └───Release
├───cryptor_dll
│   ├───antihooks
│   ├───api
│   ├───chacha20
│   ├───Debug
│   ├───filesystem
│   ├───global
```

```
│   ├───logs
│   ├───network_scanner
│   ├───obfuscation
│   ├───Release
│   └───x64
│       ├───Debug
├───Debug
├───decryptor
│   ├───chacha20
│   ├───Debug
│   ├───filesystem
│   ├───global
│   ├───network_scanner
│   ├───obfuscation
│   ├───Release
│   ├───threadpool
│   └───x64
│       └───Release
└───x64
```

This is a fairly large project with many components. There are many dozens of files containing the actual C code that are not shown in the above listing.

This is ransomware. Ransomware typically includes two components:

- Encryptor: The code that locks files on the system using encryption, making them unusable.
- Decryptor: The code that reverses the encryption, usually provided after ransom payment.

In this solution, we have the following components which reflect this:

- `decryptor` located at decryptor\decryptor.vcxproj
- `cryptor` located at cryptor\cryptor.vcxproj
- `cryptor_dll` located at cryptor_dll\cryptor_dll.vcxproj

If you look at `conti_v3.sln`, you can see over a dozen configurations for x86, x64, debug, and release versions of the ransomware tools.

Cryptor Build Process

For now, let's look at the `cryptor/cryptor.vcxproj` file.

This file designates a single compilable executable project called `cryptor` which includes many `.c` and `.h` files to be built using MSBuild version 15. Of particular interest is the following block:

```
<ItemDefinitionGroup Condition="'$(Configuration)|$(Platform)'=='Release|x64'">
  <ClCompile>
    <WarningLevel>Level3</WarningLevel>
    <Optimization>MaxSpeed</Optimization>
    <FunctionLevelLinking>true</FunctionLevelLinking>
    <IntrinsicFunctions>true</IntrinsicFunctions>
    <SDLCheck>true</SDLCheck>
```

```
      <ConformanceMode>true</ConformanceMode>
      <PositionIndependentCode>false</PositionIndependentCode>
      <DisableSpecificWarnings>4996;%(DisableSpecificWarnings)
</DisableSpecificWarnings>
      <AdditionalOptions>-mllvm -sub -mllvm -sub_loop=3 -mllvm -bcf -mllvm -
bcf_loop=3 -mllvm -bcf_prob=40 -mllvm -fla -mllvm -split -mllvm -split_num=3 %
(AdditionalOptions)</AdditionalOptions>
      <ExceptionHandling>false</ExceptionHandling>
      <RuntimeLibrary>MultiThreaded</RuntimeLibrary>
      <CppLanguageStandard>c++1y</CppLanguageStandard>
      <MSCompatibility>true</MSCompatibility>
      <MSCompatibilityVersion>1913</MSCompatibilityVersion>
      <CompileAs>CompileAsCpp</CompileAs>
      <AssemblerOutput>All</AssemblerOutput>
    </ClCompile>
    <Link>
      <SubSystem>Windows</SubSystem>
      <EnableCOMDATFolding>true</EnableCOMDATFolding>
      <OptimizeReferences>true</OptimizeReferences>
      <GenerateDebugInformation>false</GenerateDebugInformation>
      <AssemblyDebug>false</AssemblyDebug>
    </Link>
  </ItemDefinitionGroup>
```

Notice that while we are using the Visual Studio MSBuild build tool, the actual compilation toolchain in use is LLVM:

```
<AdditionalOptions>
    -mllvm -sub -mllvm -sub_loop=3 -mllvm -bcf -mllvm -bcf_loop=3 -mllvm -
bcf_prob=40 -mllvm -fla -mllvm -split -mllvm -split_num=3 %(AdditionalOptions)
</AdditionalOptions>
```

These options indicate the use of a specific version of the LLVM compiler which is capable of obfuscation. It is possible they are using directly the obfuscating LLVM found here `https://github.com/obfuscator-llvm/obfuscator`, or are using a custom or forked version. There's no way to say for sure from this information, as the leak did not include the toolchain executables.

Here's a detailed explanation of each LLVM option used:

Option	Description
`-mllvm`	This is a way to pass custom options directly to the LLVM backend. It's a common prefix for LLVM-specific options when using a compiler like clang.
`-sub`	Stands for "Substitution Pass". This obfuscation technique replaces instructions with equivalent but more complex sequences. It helps in making the code less readable and harder to reverse engineer.

Option	Description
`-sub_loop=3`	Specifies the number of iterations for the substitution pass. The value `3` means the substitution pass will be applied three times to increase the complexity of the generated code.
`-bcf`	Stands for "Bogus Control Flow". This technique introduces fake control flow in the code, making it harder for someone analyzing the code to understand its logic.
`-bcf_loop=3`	Sets the number of times the bogus control flow pass is applied. Here, it is applied three times to further obfuscate the control flow.
`-bcf_prob=40`	Specifies the probability (in percentage) that a control flow instruction will be obfuscated. A value of `40` means there's a 40% chance for each control flow instruction to be altered by the bogus control flow technique.
`-fla`	Stands for "Flattening". This technique flattens the control flow graph of functions. It transforms the code's structure to a more linear sequence with artificial control flow added, making the program flow harder to follow.
`-split`	Refers to "Instruction Splitting". This technique splits instructions into multiple simpler instructions, increasing the code size and complexity, thus making reverse engineering more difficult.
`-split_num=3`	Indicates the number of times the instruction splitting should be applied. The value `3` means each instruction will be split into three separate instructions.

Interestingly, the compiler frontend is not declared. It's possible this is being overridden at the global level on the Conti build machine.

Runtime Evasion Techniques

If we investigate the source code, we can see some evasion techniques that are built into the code, aside from the compiler level obfuscation:

```
// In `cryptor/antihooks/antihoops.cpp`

HMODULE hKernel32 = apLoadLibraryA(OBFA("kernel32.dll"));

...

if (hKernel32) {
    removeHooks(hKernel32);
}
```

Here, we see a function that resolves the address of the WinAPI function `LoadLibrary` in an obfuscated manner:

```
// In `cryptor/api/getapi.h`
__forceinline
```

```
HMODULE
WINAPI
apLoadLibraryA(
    LPCSTR lpLibFileName
)
{
    HMODULE(WINAPI * pFunction)(LPCSTR);
    pFunction = (HMODULE(WINAPI*)(LPCSTR))getapi::GetProcAddressEx2(NULL,
KERNEL32_MODULE_ID, 0x439c7e33, 4);
    return pFunction(lpLibFileName);
}
```

The `removeHooks` function is too large to list here, but at a high level, this function will do the following:

1. Access the currently loaded copy of the library
2. Open the version on disk
3. Compare the exported functions in each copy with a simple memory comparison function
4. If the function is hooked, overwrite the hooked version with the version from the file on disk

Encryption Functionality

Inside main, we see use of a `threadpool` class. The actual encryption is nested within a few function calls from in here.

The ransomware uses a multithreading approach to speed up encryption. Otherwise, the ransom process would take an extremely long time.

The threadpool function that actually triggers encryption is `ThreadPoolHandler`. This function will call another function which will trigger the encryption of a specific file.

Issues with The Conti Build Approach

For such a prolific ransomware, the source code reveals many issues.

First, code is rewritten in several places. Many core functionalities, such as the multithreading code, are re-written in all three projects. This is curious, as the developers are certainly aware of the purpose of libraries.

Second, the developers are likely relying on global-level build configurations in order to use the `clang` compiler frontend. This choice suggests an ad-hoc approach to the development of this solution. The build would not be directly migrate-able to a different system. Ideally, the compiler would be set at both the project and global level.

Finally, the `vcxproj` files show that the obfuscation passes are applied in all build versions, even debug builds. This would make debugging the code much more difficult. This might not be a problem for such a simple project, but for more advanced code development, debugging is essential.

The odd configuration of the `vcxproj` (with regard to the compiler frontend) is likely a result of difficulties integrating the LLVM compilation toolchain into Visual Studio. With ordinary LLVM, a developer would select LLVM-Clang as the platform toolset in project properties. Using off-the-shelf obfuscating LLVM can require a custom platform toolset, which is not straightforward to create, or careful drop-in of modified LLVM components.

Resources on OLLVM Integration with VS

- https://ghoulsec.medium.com/reddev-series-4-experimenting-syswhisper2-with-llvm-obfuscator-fa4dd39df2da

Safety Issues with Unvetted Toolchain Components

Many forum and blog posts (see Medium, UnknownCheats) demonstrate drop-in replacement of normal LLVM platform toolset components in Visual Studio. It should be noted that doing so represents a supply-chain attack risk, especially with some of the more random Obfuscator LLVM variants on the web. Backdoors in toolchain components are very much possible, and without strict vetting of the source tree, it is impossible to rule out. For professional Red Teams (which is what this series is intended for), the risk is unacceptable. The best approach would be to develop a custom version based on a reliable copy of the actual LLVM source tree. Obfuscation passes can be implemented manually with enough prerequisite knowledge. Ransomware groups may have the luxury of not caring if their "clients" are backdoored by someone else, but we as Red Teamers must protect our clients from unnecessary risks.

Case Study - Frank2's Packer

Another excellent example of a malware development project with a comprehensive toolchain is Frank2's Packer Tutorial. This is a *packer* project utilizing CMake as the build system. The repo includes a detailed tutorial.

From the guide:

> A packer is a program that decompresses and launches another program within its address space (or sometimes, another process's address space). It is sometimes known for being the vector that attacks analysis environments, such as debuggers and virtual sandboxes.

This means we have some complicated build steps due to *dependencies*. The packer (software that performs the packing of one executable into another) is dependent on the stub component. The stub is what is eventually deployed to the target system and contains the compressed and encrypted payload. This means we must package the stub executable and the packer together for the packer to work.

Another point of dependency is `zlib`. `zlib` is a compression library that is used in this project for the compression component.

The readme for the project explains the whole process completely, so I won't repeat that here.

Case Study - Post Exploitation Frameworks

Post exploitation frameworks (such as Cobalt Strike) use various methods for generating payload executables. These frameworks tend to use methods that are even more automated than a traditional build system, as users demand the ability to just generate a payload with a single command / button click.

Cobalt Strike & Meterpreter

CS ships with default payloads for each implant variant provided. These raw payloads are then patched with customization options. By default, CS will embed the raw payloads into prebuilt executables for DLLs, service binaries, and plain executables. Additionally, a default reflective loader is shipped with these, which may be

modified (called a *user defined reflective loader* or UDRL). The way all this works (and which templates are used) can be configured using "Malleable C2" profiles and Aggressor scripts.

Metasploit's default implant also works in a similar way through `msfvenom`, although `msfvenom` offers many obfuscation and encoding options for the output payload.

Sliver

Sliver, another popular implantation framework, uses a different approach. The implant is written in Go, and is generated with custom user options using Go's templating engine. This makes the codebase very unwieldy, but it does work. Once the source code has been generated, it will be run through `gobfuscate` in order to obfuscate symbols in the output binary.