

Forum:[Hack Forums](#)

Thread: [\[C++\] Let's write a code mutation engine: Part 0: Preface and Setup](#)

binbus - October 4, 2019, 9:39 pm

Over the past week, I've written a code mutation engine.

In this tutorial series, I will teach you how to write your own x86 code mutation engine, purely for educational purposes, based on my own engine. This code shall not be used for any illegal purposes whatsoever. It is only posted here so that users interested in computer security and compilers can see how neat these things can be, and so that users may learn more about computer science

This tutorial series is for people with basic knowledge of C++, algorithms & data structures, x86 assembly, basics of disassembly, and compiler theory basics

Once the engine is built, it can trivially be expanded to support arbitrary x86 PE files and the mutation engine can be extended to support more instructions.

Our engine will take a binary file consisting of a sequence of x86 instructions and output mutated code

Tutorial releases are contingent upon popularity. I will stop this tutorial series if there is not enough participation or interest.

I use the following libraries in this tutorial series with Microsoft C++ compiler.
Boost for parsing command line inputs

Capstone for disassembly engine

Keystone for assembly engine

Part 1: Linear Disassembly:
Read our input code

Use a linear disassembly algorithm to disassemble our code and represent it in memory

Represent our assembly into basic block structures

Display our disassembly in console

Part 2: Control Flow Flattening:
Create a control flow stub which will handle all of the control flow in our program

Redirect all basic block transfer and next flows to our control flow stub

Rearrange our program (randomly) in memory

Save our program to disk

Part 3: Junk Code Insertion:

Create a junk code engine and mutate our binary

Part 4: Code Mutation: Variable Instruction Substitution and Peephole (De)Optimization
Substitute instructions for other instructions which can be of different lengths

Use peephole (de)optimization to substitute instruction chunks

Apply techniques to our input randomly

Part 5: Instruction Selection (De)Optimization:

Learn about data-flow analysis techniques for live variable sets

Create a data-flow analysis engine using capstone to explain live variable sets within basic blocks

Rearrange instructions within basic blocks in memory

Part 6: Register Allocation Mutation:

Convert input x86 code to our own intermediate representation language using our data-flow engine from tutorial 5

Create a data-flow engine for our intermediate representation language

Compute our (random) register allocations using linear scan algorithm

Our engine will mutate code by using parts 1-6 to itself, using randomization, and continued re-application.

To keep you guys interested I'm going to preface this tutorial with results of tutorial 2, which is just two tutorials away.

You will be able to do this yourself shortly

I take the following C code, which is written just as a toy example to have enough basic blocks in it:

PHP Code

```
<?php
int sumParity(int list[], int size, int* even, int* odd)
{
```

```

int sumEven = 0;
int sumOdd = 0;

for (int i = 0; i < size; i++)
{
    if (list[i] % 2 == 0)
    {
        sumEven += list[i];
    }
    else
    {
        sumOdd += list[i];
    }
}

*even = sumEven;
*odd = sumOdd;

return sumEven + sumOdd;
}

```

Using gcc, I compile it and display it to you with IDA:

Entry: 0x00

[\[ASM \(NASM\)\] seg000:00000000 seg000 segment byte public 'CODE' use32 seg000:00 - Pastebin.com](#)

I now show you the results of applying our code flattening engine (without any mutation) with IDA:

Entry: 0xA4

[\[ASM \(NASM\)\] seg000:00000000 ; Segment type: Pure code seg000:00000000 seg000 segme - Pastebin.com](#)

I now show you the results of applying our code flattening engine twice (without any mutation) with IDA:

Entry: 0x1B9

[\[ASM \(NASM\)\] seg000:00000000 seg000 segment byte public 'CODE' use32 seg000:00000000 - Pastebin.com](#)

You will be able to do the same after tutorial 2

Setting Up Your Environment

Download and install Visual Studio 2019 Community

Download and install cmake

Download capstone library from [The Ultimate Disassembly Framework Capstone The Ultimate Disassembler](#)

Download keystone library from [Keystone The Ultimate Assembler](#)

Compile and build capstone as a static library for both debug and release builds x86 using cmake. Use cmake to generate Visual Studio project files for capstone. In the capstone configuration file, make sure you compile for the x86 architecture, do not enable capstone diet, enable capstone

reduce, enable capstone static library.

Compile and build keystone as a static library for both debug and release builds x86 using cmake

Follow the capstone and keystone tutorials here and make sure you can build a project using these two libraries

Capstone tutorial: [Programming with C language Capstone The Ultimate Disassembler](#)

Keystone tutorial: [Programming with C & Python Keystone The Ultimate Assembler](#)

Build a project with boost and read up on boost program options, you can add it to your Visual Studio project by using NuGet by adding "boost_program_options-src" to your NuGet package manager. (Google usage of NuGet with C++. It can be done solely via GUI).

SKIDROOTKIT - *October 4, 2019, 9:41 pm*

OP is very intelligent. Good post.

malpwn - *October 4, 2019, 10:35 pm*

(Oct 4, 2019 09:39 PM)binbus Wrote: [\[CITE\]](#) Over the past week, I've written a code mutation engine.

In this tutorial series, I will teach you how to write your own x86 code mutation engine, purely for educational purposes, based on my own engine. This code shall not be used for any illegal purposes whatsoever. It is only posted here so that users interested in computer security and compilers can see how neat these things can be, and so that users may learn more about computer science

This tutorial series is for people with basic knowledge of C++, algorithms & data structures, x86 assembly, basics of disassembly, and compiler theory basics

Once the engine is built, it can trivially be expanded to support arbitrary x86 PE files and the mutation engine can be extended to support more instructions.

Our engine will take a binary file consisting of a sequence of x86 instructions and output mutated code

Tutorial releases are contingent upon popularity. I will stop this tutorial series if there is not enough participation or interest.

I use the following libraries in this tutorial series with Microsoft C++ compiler.
Boost for parsing command line inputs

Capstone for disassembly engine

Keystone for assembly engine

Part 1: Linear Disassembly:
Read our input code

Use a linear disassembly algorithm to disassemble our code and represent it in memory

Represent our assembly into basic block structures

Display our disassembly in console

Part 2: Control Flow Flattening:

Create a control flow stub which will handle all of the control flow in our program

Redirect all basic block transfer and next flows to our control flow stub

Rearrange our program (randomly) in memory

Save our program to disk

Part 3: Junk Code Insertion:

Create a junk code engine and mutate our binary

Part 4: Code Mutation: Variable Instruction Substitution and Peephole (De)Optimization

Substitute instructions for other instructions which can be of different lengths

Use peephole (de)optimization to substitute instruction chunks

Apply techniques to our input randomly

Part 5: Instruction Selection (De)Optimization:

Learn about data-flow analysis techniques for live variable sets

Create a data-flow analysis engine using capstone to explain live variable sets within basic blocks

Rearrange instructions within basic blocks in memory

Part 6: Register Allocation Mutation:

Convert input x86 code to our own intermediate representation language using our data-flow engine from tutorial 5

Create a data-flow engine for our intermediate representation language

Compute our (random) register allocations using linear scan algorithm

Our engine will mutate code by using parts 1-6 to itself, using randomization, and continued re-application.

To keep you guys interested I'm going to preface this tutorial with results of tutorial 2, which is just two tutorials away.

You will be able to do this yourself shortly

I take the following C code, which is written just as a toy example to have enough basic blocks in it:

PHP Code

```
<?php
int sumParity(int list[], int size, int* even, int* odd)
{
    int sumEven = 0;
    int sumOdd = 0;

    for (int i = 0; i < size; i++)
    {
        if (list[i] % 2 == 0)
        {
            sumEven += list[i];
        }
        else
        {
            sumOdd += list[i];
        }
    }

    *even = sumEven;
    *odd = sumOdd;

    return sumEven + sumOdd;
}
```

Using gcc, I compile it and display it to you with IDA:

Entry: 0x00

[\[ASM \(NASM\)\] seg000:00000000 seg000 segment byte public 'CODE' use32 seg000:00 - Pastebin.com](#)

I now show you the results of applying our code flattening engine (without any mutation) with IDA:

Entry: 0xA4

[\[ASM \(NASM\)\] seg000:00000000 ; Segment type: Pure code seg000:00000000 seg000 segme - Pastebin.com](#)

I now show you the results of applying our code flattening engine twice (without any mutation) with IDA:

Entry: 0x1B9

[\[ASM \(NASM\)\] seg000:00000000 seg000 segment byte public 'CODE' use32 seg000:00000000 - Pastebin.com](#)

You will be able to do the same after tutorial 2

Setting Up Your Environment

Download and install Visual Studio 2019 Community

Download and install cmake

Download capstone library from [The Ultimate Disassembly Framework Capstone The Ultimate Disassembler](#)

Download keystone library from [Keystone The Ultimate Assembler](#)

Compile and build capstone as a static library for both debug and release builds x86 using cmake. Use cmake to generate Visual Studio project files for capstone. In the capstone configuration file, make sure you compile for the x86 architecture, do not enable capstone diet, enable capstone reduce, enable capstone static library.

Compile and build keystone as a static library for both debug and release builds x86 using cmake

Follow the capstone and keystone tutorials here and make sure you can build a project using these two libraries

Capstone tutorial: [Programming with C language Capstone The Ultimate Disassembler](#)

Keystone tutorial: [Programming with C & Python Keystone The Ultimate Assembler](#)

Build a project with boost and read up on boost program options, you can add it to your Visual Studio project by using NuGet by adding "boost_program_options-src" to your NuGet package manager. (Google usage of NuGet with C++. It can be done solely via GUI).

Everything is kool and the gang but Boost for such (not) so trivial task ?

binbus - October 4, 2019, 10:44 pm

(Oct 4, 2019 10:35 PM)malpwn Wrote: [\[CITE\]](#) Everything is kool and the gang but Boost for such (not) so trivial task ?

No point re-inventing the wheel. Boost program options will be used to parse command line options to our engine.

tux1 - October 5, 2019, 2:21 am

Looks interesting, can't wait to see the next part!

binbus - October 27, 2019, 7:14 pm

There won't be a next part. I've received a few DM's demanding the source code (so they could resell it) and one instance one of them even negged me for denying him.

Enjoy your shithole forum

YouthInAsia - October 28, 2019, 5:13 am

(Oct 27, 2019 07:14 PM)binbus Wrote: [\[CITE\]](#) There won't be a next part. I've received a few DM's demanding the source code (so they could resell it) and one instance one of them even negged me for denying him.

Enjoy your shithole forum

Imagine releasing anything on trashforums.net and not expecting it to be recycled and sold as "NEW FUD C++ CRYPTOR" it's why I don't care to help people on this forum anymore it's absolute shit.