Acquiring .NET Objects from the Managed Heap

Topher Timzen
Dept. of Computer Science
Southern Oregon University
2015

Abstract

This paper will describe how to use any instantiated object in the .NET CLR managed heap as if it were declared locally. It will be shown that by referencing object pointers from the managed heap, an attacker control objects being used in an application. Reflective techniques will be discussed and a signature will be introduced to find any object on the managed heap.

I. Introduction

The .NET Framework uses the Common Language Runtime, CLR, to manage the execution of .NET programs. There are 4 major versions of the CLR available and multiple .NET versions within each [2].

CLR Version	.NET Version
1.0	1.0
1.1	1.1
2.0	2.0, 3.0, 3.5
4.0	4.0, 4.5

For the purpose of this paper, only CLR versions 2.0 and 4.0 will be discussed as the authors feels 1.0 and 1.1 are not relevant for attacking modern applications.

Using Reflection [4] key information about an object can be discovered. A plethora of information in objects is useful for attacking or reverse engineering such as a list of fields, instance methods, static methods and variables that an object contains. Furthermore, once an object is locally accessible an attacker can manipulate it in any way they want.

Any object can be instantiated using reflection. While that local reference is not necessarily usable by an attacker, it can be used to reveal both the signature of all objects of that class and the memory location of the managed heap, bypassing any address space layout randomization, ASLR. All instantiated objects from a specific class share the same Method Table (MT) pointer, which will be used later to locate objects on the managed heap.

Once the Method Table for a specific class is known and the managed heap discovered, an attacker can scan through the managed heap to locate all objects instantiated from a specific class and use them locally. To showcase this technique a sample application written in .NET will be utilized and compiled in x86 and x64 with both the 2.0 and 4.0 CLR.

II. Finding objects with WinDbg

In order to understand what objects look like in memory WinDbg was used frequently in the author's analysis. Using the SOS Debugging Extension (SOS.dll), WinDbg allows the user to debug managed applications (programs using the CLR)[3]. The most useful extensions to locate and understand objects are *dumpheap* and *dumpobj*. The following WinDbg results are for the 2.0 CLR in x86, but the same techniques apply for all runtimes and x64.

Running *dumpheap* will show all of the objects on the managed heap, where each object lives on the heap, the size of the object and its MT pointer. Most of the addresses from *dumpheap* for the following example are within the range 0x02721000 to 0x0284f35c with a few appearing from 0x03721000 to 0x037f0578. So far, objects of concern have appeared within the lesser range as newly created objects are allocated there.

```
0:009>!dumpheap
Address   MT              Size
[snip]
0284c1b8 6e3da388        36
0284c1dc 6b2b390c        72
0284c224 6b2b3ba0        16
0284c234 6e3da4f8        16
0284c244 6e3d9fb4        32
0284c264 6b2b3ba0        16
[snip]
total 25237 objects
```

Furthermore, this extension will show the statistics for each object instantiated which includes the number of them present on the managed heap.

```
0:009>!dumpheap
[snip]
Statistics:
    MT    Count    TotalSize Class Name
[snip]
6db6ce80        1          100 System.Diagnostics.FileVersionInfo
6b2aa3e4        5          100 System.Windows.Forms.RichTextBox+OleCallback
6af488e8        5          100 System.Configuration.ConfigurationSchemaErrors
007c078c        5          100 GrayStorm.shellcode.dataBox
```

All objects of the same type share the same Method Table, which provides metadata about Object Instances [1]. Using the *-mt* parameter on *dumpheap* with a MT address from above, all objects using that Method Table are shown.

```
0:008> !dumpheap -mt 007c078c
Address       MT     Size
027fb5b0 007c078c       20
027fb5e4 007c078c       20
027fb5f8 007c078c       20
027fb60c 007c078c       20
027fbf84 007c078c       20
total 5 objects
Statistics:
 MT       Count TotalSize Class Name
007c078c 5       100 GrayStorm.shellcode.dataBox
Total 5 objects
```

Now all objects instantiated from GrayStorm.shellcode.dataBox are shown as they all pointed back to the MT at 0x007c078c. *dumpobj* can now be used on the address of an object to show its fields and more information about it.

```
0:008> !dumpobj 027fb5b0
Name: GrayFrost.shellcode.dataBox
MethodTable: 007c078c
EEClass: 00336fd4
Size: 20(0x14) bytes
 (C:\bin\GrayFrost.exe)
Fields:
 MT       Field   Offset  Type           VT Attr    Value    Name
6e3e0d48  40001ac  4       System.String  0 instance 027fb45c name
6e3e37b8  40001ad  8       System.Byte[]  0 instance 027fae0c data
6e3e2f94  40001ae  c       System.Int32   1 instance 0 indexToStartCleaning
```

The first four bytes of the above object hold a pointer back to its Method Table.

```
0:008> db 027fb5b0
027fb5b0  8c 07 7c 00 5c b4 7f 02-0c ae 7f 02 00 00
    00 00    ..|.\...........
```

Reconstructing the whole Method Table and Object Instance is not important because once a reference object is available in a local scope reflection can be used and anything about that object can be seen. The size of an Object Instance is necessary to create a robust scanner and the size is the second four byte block of a Method Table as shown by Kommalapati and Christian [1].

Using the knowledge that an instance object's first four bytes are the Method Table and all like objects share the same pointer, an attacker can find the location of the heap in memory. Once the heap is located they can brute force it by looking for the Method Table reference they require to obtain object references.

III. Finding Objects at runtime

In order to find objects at runtime, the exact location of the managed heap needs to be discovered. To discover this location a signature for the specific kind of object for which one is searching needs to be instantiated. Once instantiated, the managed heap location can be found as well as the MT for the object. Utilizing Reflection the constructor of a class can be called to instantiate a local reference.

```
Type refc = typeof(GrayFrost.testMethods);
ConstructorInfo ctor = refc.GetConstructor      (Type
    .EmptyTypes);
object wantedObject = ctor.Invoke(new object[]{});
```

Once we have a local reference we are able to discover its raw memory address by manipulating a method's stack frame (Keep in mind that the details of obtaining the raw IntPtr to an object differs between x86 and x64 assembly because of how they handle argument passing). For both architectures, unsafe code will be utilized in C# (which is still usable in a target application compiled disallowing unsafe code) to obtain a raw object pointer.

A. x86

In x86 architecture arguments are pushed onto the stack in reverse order. In order to obtain the object pointer, an IntPtr will be declared locally and then dereferenced to obtain the objects pointer in memory by traversing through the current stack frame.

```
public static IntPtr getObjectAddr(object
    wantedObject)
{
  IntPtr objPtr = IntPtr.Zero;
  unsafe
  {
    objPtr = *(&objPtr - 3);
  }
  return objPtr; //0x260a4c8
}
```

objPtr will now contain the address of the wantedObject. In the .NET CLR object instances are pointers back to their Object Table on the managed heap, which means we now know the location of the heap.

```
0:008> !do 260a4c8
Name: GrayFrost.testMethods
MethodTable: 00286d34
EEClass: 00382348
Size: 12(0xc) bytes
(C:\bin\GrayFrost.exe)
Fields:
MT       Field   Offset Type          VT Attr   Value    Name
6e0437b8 4000002 8      System.Byte[] 0  static 0260a3fc objectPtr
```

Now that the address of the Object Table is known and there is a reference to the Method Table location, the first four bytes of memory from the object location at 0x260a4c8, the managed heap can be brute-forced for other objects matching that signature. The below pseudocode is the author's approach to brute forcing the managed heap. For searching at a negative offset the size field (from the MT) cannot be utilized and the addresses are read linearly.

```
While valid memory at positive offset from object
  Obtain object size and jump to next object
  Check first four bytes for matching MethodTable
  IF MethodTables match
    Add object IntPtr to list
While valid memory at negative offset from object
  Check each 4 byte MT address to see if its address
       is the same as the wantedObjects
  IF MethodTables match
    Add object IntPtr to list
```

Once the brute forcing is finished a listing of all object IntPtrs of GrayFrost.testMethods are present and need to be converted back into the object type. Again utilizing stack manipulation .NET can be tricked into placing an IntPtr into an object pointer because as previously shown objects are IntPtrs. The below code will take an IntPtr and place it into a local object.

```
public static object GetInstance(IntPtr ptrIN)
{
  object refer = ptrIN.GetType();
  IntPtr objPtr = ptrIN;
  unsafe
  {
    *(&objPtr - clrSub) = *(&objPtr);
  }
  return refer;
}
```

1) CLR 2.0 vs CLR 4.0: I discovered for both the 2.0 and 4.0 CLR on x86 the wantedObject parameter was at a negative stack offset of 3 from objPtr. Also, the clrSub offset is 1 for CLR 2.0 and 2 for CLR 4.0 to place an IntPtr into an object.

B. x64

By placing three local variables in a method, see below, I discovered that the address of the object will become present. If there are less than three local arguments, the CLR does not place the object pointer in a reachable range.

```
public static IntPtr getObjectAddr64(object
    wantedObject)
{
  IntPtr objPtr = (IntPtr)4;
  object refer = wantedObject;
  IntPtr objPtr2 = (IntPtr)8;
  unsafe
  {
    objPtr = *(&objPtr + clrSub);
  }
  return objPtr;
}
```

Once the objPtr is known, the x86 pseudo code for finding objects will also work. For getting an object back from an IntPtr, the below code can be used.

```
public static object GetInstance64(IntPtr
    wantedObject)
{
  IntPtr objPtr = wantedObject;
  object refer = wantedObject.GetType();
  IntPtr objPtr2 = (IntPtr)8;
  unsafe
  {
    *(&objPtr + 1) = *(&objPtr);
  }
  return refer;
}
```

1) CLR 2.0 vs CLR 4.0: There are no differences in the above methods for CLR 2.0 and 4.0 for x64 assembly.

IV. Conclusion

An attacker now has the ability to instantiate any object of their choice, brute-force the managed heap for other objects instantiated from the same class, and use them as if they were declared locally. Using Reflection, all of the fields and properties of an object can be viewed and altered and instance methods can be called. The underlying power for .NET attack chains using constructed objects is massive as attackers can instantiate any object, find all other objects on the heap that match it and do anything with them.

References

[1] Hanu Kommalapati and Tom Christian, *Drill Into .NET Framework Internals*.
[2] Microsoft Corporation. Common Language Runtime (CLR).
[3] Microsoft Corporation. SOS.dll (SOS Debugging Extension)
[4] Jon. Reflections Hidden Power. May 2002.