

Locreate: An Anagram for Relocate

12/2006

skape
mmiller@hick.org

Chapter 1

Foreword

Abstract: This paper presents a proof of concept executable packer that does not use any custom code to unpack binaries at execution time. This is different from typical packers which generally rely on packed executables containing code that is used to perform the inverse of the packing operation at runtime. Instead of depending on custom code, the technique described in this paper uses documented behavior of the dynamic loader as a mechanism for performing the unpacking operation. This difference can make binaries packed using this technique more difficult to signature and analyze, but only when presented to an untrained eye. The description of this technique is meant to be an example of a fun thought exercise and not as some sort of revolutionary packer. In fact, it's been used in the virus world many years prior to this paper.

Thanks: The author would like to thank Skywing, spoonm, deft, intropy, Orlando Padilla, nemo, Richard Johnson, Rolf Rolles, Derek Soeder, and Andre Protas for their discussions and feedback.

Challenge: Prior to reading this paper, the author recommends that the reader attempt to determine the behavior of the packer that was used on the binary included in the attached code sample. The binary itself is innocuous and just performs a few simple printf operations.

Previous Research: This technique has been used in the virus world far in advance of this writing. Examples that apply this technique include W95/Resurrel and W95/Silcer. Further research indicates that Peter Szor did a write-up on this technique entitled "Tricky Relocations" in the April 2001 edition of *Virus Bulletin* [2, 3].

Chapter 2

Locreate

Executable packers, such as UPX, are commonly employed by malware as a means of delaying or otherwise thwarting the process of static analysis. Packers also have perfectly legitimate uses, but these uses fall outside of the scope of this paper. The reason packers make static analysis more difficult is because they alter the form of the binary to the point that what appears on disk is entirely different from what actually ends up executing in memory. This alteration is typically accomplished by encapsulating a pre-existing binary in a “host” binary. The algorithm used to encapsulate the pre-existing binary in the host binary is what differs from one packer to the next. In most cases, the host binary must contain code that will perform the inverse of the packing operation in order to decapsulate the original binary. The code that is responsible for performing this operation is typically referred to as an *unpacker*. The process of unpacking the original binary is usually done entirely in memory without writing the original version out to disk. Once the original binary is unpacked, execution control is transferred to the original binary which begins executing as if nothing had changed.

This general approach represents an easy way of altering the form of a binary without changing its effective behavior. In fact, it’s pretty much analagous to payload encoders that are used in conjunction with exploits to alter the form of a payload in order to satisfy some character restrictions without changing the payload’s effective behavior. In the case of payload encoders, some arbitrary code must be prefixed to the encoded payload in order to perform the inverse of the encoding operation once the payload is executed. However, like payload encoders, the use of custom code to perform the inverse of the packing or encoding operation can lead to a few problems.

The most apparent of these problems has to do with the fact that while the packed form of an executable may be entirely different from its original, the

code used to perform the unpacking operation may be static. In the event that the unpacker consists of static code, either in whole or in part, it may be possible to signature or otherwise identify that a particular packing algorithm has been used to produce a binary and thus make it easier to restore the original form of the binary. This ability is especially important when it comes to attempting to heuristically identify malware prior to allowing a user to execute it.

The use of custom code can also make it possible for tools to be developed that attempt to identify unpackers based on their behavior. Ero Carrera has provided some excellent illustrations relating to the feasibility of this type of attack against unpackers[1]. An understanding of an unpacker's behavior may also make it possible to acquire the original binary without allowing it to actually execute by simply tracing the unpacker up until the point where it transfers execution control to the original binary. In the case of malware, this weakness means that benefits gained from packing an executable can be completely nullified.

Both of these problems are meant to illustrate that even though custom unpacking code is often a requirement, its mere presence exposes a potential point of weakness. If it were possible to eliminate the custom code required to unpack a binary, it could make the two problems described previously much more difficult to realize. To that point, the technique described in this paper does not rely on the presence of custom code in a packed binary in order to unpack itself. Instead, documented behavior of the dynamic loader is used to perform the unpacking whenever the packed binary is executed. While this approach has its benefits, there are a number of problems with it that will be discussed later on. In the interest of brevity, the packer described in this paper will simply be referred to as *locreate*. As was already mentioned, *locreate* leverages a documented feature of most dynamic loaders in order to perform its unpacking operation. Given that the process of unpacking typically involves transforming the original binary's contents back into its original form, there are only a finite number of dynamic loader features that might be abused. Perhaps the feature that is best suited for transforming the contents of a binary at runtime is the dynamic loader feature that was designed to do just that: relocations.

In the event that a binary is unable to be loaded at its preferred base address at runtime, the dynamic loader is responsible for attempting to move the binary to another location in memory. The act of moving a binary from its preferred base address to a new base address is more commonly referred to as *relocating*. When a binary is relocated to a new base address, any references the binary might have to addresses that are relative to its preferred base address will no longer be valid. As such, references that are relative to the preferred base address must be updated by the dynamic loader in order to make them relative to the new base address. Of course, this presupposes that the dynamic loader has some knowledge of where in the binary these address references are made. To satisfy this presupposition, binaries will typically include *relocation information* to provide the dynamic loader with a map to the locations within the binary that

need to be adjusted. When a binary does not include relocation information, it's classified as a *non-relocatable binary*. Without relocation information, a binary cannot be relocated to an alternate base address in an elegant manner (ignoring position independent executables).

The structures used to convey relocation information differs from one binary format to the next. For the purpose of this paper, only the structures used to describe relocations of Portable Executable (PE) binaries will be discussed. However, it should be noted that the approaches described in this paper should be equally applicable to other binary formats, such as ELF¹. The PE binary format conveys relocation information through one of the data directories that is included within the optional header portion of the NT header. This data directory is symbolically referred to through the use of the `IMAGE_DIRECTORY_ENTRY_BASERELOC`. The base relocation data directory consists of zero or more `IMAGE_BASE_RELOCATION` structures which are defined as:

```
typedef struct _IMAGE_BASE_RELOCATION {
    ULONG   VirtualAddress;
    ULONG   SizeOfBlock;
    // USHORT TypeOffset[1];
} IMAGE_BASE_RELOCATION, *PIMAGE_BASE_RELOCATION;
```

The base relocation data directory is a little bit different from most other data directories. The `IMAGE_BASE_RELOCATION` structures embedded in the data directory do not occur immediately one after the other. Instead, there are a variable number of `USHORT` sized fixup descriptors that separate each structure. The `SizeOfBlock` attribute of each structure describes the entire size of a relocation block. Each relocation block consists of the base relocation structure and the variable number of fixup descriptors. Therefore, enumeration of the base relocation data directory is best performed by using the `SizeOfBlock` attribute of each structure to proceed to the next relocation block until none are remaining. The `VirtualAddress` attribute of each relocation block is a page-aligned *relative virtual address* (RVA) that is used as the base address when processing its associated fixup descriptors. In this manner, each relocation block describes the relocations that should be applied to exactly one page.

The fixup descriptors contained within a relocation block describe the address of the value that should be transformed and the method that should be used to transform it. The PE format describes about 10 different transformations that can be used to fixup an address reference. These transformations are conveyed through the top 4 bits of each fixup descriptor. The bottom 12 bits are used to describe the offset into the `VirtualAddress` of the containing relocation block. Adding the bottom 12 bits of a fixup descriptor to the `VirtualAddress`

¹In fact, other binary formats make the technique used by `locreate` even easier. For example, ELF supports applying relocation fixups with an addend. This addend is basically an arbitrary value that is used in conjunction with a transformation.

of a relocation block produces the RVA that contains a value that needs to be transformed. Of the transformation methods that exist, the one most commonly used on x86 is `IMAGE_REL_BASED_HIGHLOW`, or 3. This transformation dictates that the 32-bit displacement between the original base address and the new base address should be added to the value that exists at the RVA described by the fixup descriptor. The act of adding the displacement means that the value will be transformed to make it relative to the new base address rather than the original base address. To better understand how all of these things tie together, consider the following source code example:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World.\n");

    return 0;
}
```

When compiled down, this function appears as the following:

```
sample!main:
00401010 55          push     ebp
00401011 8bec       mov     ebp,esp
00401013 6800104200 push     offset sample!_rtc_tzz <PERF> (sample+0x21000) (00421000)
00401018 e80c000000 call    sample!printf (00401029)
0040101d 83c404     add     esp,4
00401020 33c0       xor     eax,eax
00401022 5d         pop     ebp
00401023 c3         ret
```

At address `0x00401013`, `main` pushes the address of the string that contains “Hello World!”:

```
0:000> db 00421000 L 10
00421000 48 65 6c 6c 6f 20 57 6f-72 6c 64 2e 0a 00 00 00 Hello World....
```

In this case, the `push` instruction is referring to the string using an absolute address. If the sample executable must be relocated at runtime, the dynamic loader must be provided with the relocation information necessary to fixup the reference to the absolute address. The `dumpbin.exe` utility from Visual Studio can be used to confirm that this information exists. The first requirement is that the binary must have relocation information. By default, all DLLs will contain relocation information, but executables typically do not. Executables can be compiled with relocation information by using the `/fixed:no` linker flag. When a binary is compiled with relocations, the presence of relocation

information is simply indicated by a non-zero `VirtualAddress` and `Size` for the base relocation data directory. These values can be determined through `dumpbin.exe /headers`:

```
26000 [    EE8] RVA [size] of Base Relocation Directory
```

Since relocation information must be present at runtime, there should also be a section, typically named `.reloc`, that contains the virtual mapping information for the relocation information:

```
SECTION HEADER #5
.reloc name
  1165 virtual size
  26000 virtual address (00426000 to 00427164)
  2000 size of raw data
  24000 file pointer to raw data (00024000 to 00025FFF)
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
42000040 flags
  Initialized Data
  Discardable
  Read Only
```

In order to validate that this executable contains relocation information for the absolute address reference made to the “Hello World!” string, the `dumpbin.exe /relocations` command can be used:

```
File Type: EXECUTABLE IMAGE
```

```
BASE RELOCATIONS #5
  1000 RVA,      A8 SizeOfBlock
    14 HIGHLOW      00421000
    2C HIGHLOW      00420350
...
```

This output shows the first relocation block which describes the RVA `0x1000`. Each line below the relocation block header describes the individual fixup descriptors. The information displayed includes the offset into the page, the type of transformation being performed, and the current value at that location in the binary. From the disassembly above, the location of the address reference that is being made is `0x00401014`. Therefore, the very first fixup in this relocation block provides the dynamic loader within the information necessary to change the address reference to the new base address when the binary is relocated. If this binary were to be relocated to `0x50000000`, the `HIGHLOW` transformation would be applied to `0x00401014` as follows. The displacement between the new base address and the old address would be calculated as `0x50000000`

- 0x00400000, or 0x4fc00000. Adding 0x4fc00000 to the existing value of 0x00421000 produces 0x50021000 which is subsequently stored in 0x00401014. This causes the absolute address reference to become relative to the new base address.

Based on this basic understanding of how relocations are processed, it's now possible to describe how a packer can be implemented that takes advantage of the way the dynamic loader processes relocation information. As has been illustrated above, relocation information is designed to make it possible to fixup absolute address references at runtime when a binary is relocated. These fixups are applied by taking into account the displacement between the new base address and the original base address. More often than not, this displacement isn't known ahead of time, thus making it impossible to reliably predict how the content at a specific location in the binary will be altered. But what if it were possible to deterministically know the displacement in advance? Knowing the displacement in advance would make it possible to alter various locations of the binary in a manner that would permit the original values to be restored by relocations at runtime. In effect, the on-disk version of the binary could be made to appear quite different from the in-memory version at runtime. This is the basic concept behind `locreate`.

In order for `locreate` to work it must be possible to predict the displacement reliably. Since the displacement is calculated in relation to the *preferred base address* and the *expected base address*, both values must be known. Furthermore, the binary must be relocated every time it executes in order for the relocations to be applied. As it happens, both of these problems can be solved at once. Since a binary is only guaranteed to be relocated if its preferred base address is in conflict with an existing address, a preferred base address must be selected that will always lead to a conflict. This can be accomplished by setting the preferred base address to any invalid user-mode address (any address above 0x80000000 inclusive)². Alternatively, the base address can be set to `SharedUserData` which is guaranteed to be located at 0x7ffe0000 in every process. Setting the binary's preferred base address to any of these addresses will force it to be relocated every time it executes. The only unknown is what address the binary is expected to be relocated to.

Determining the address that will be relocated to depends on the state of the process' address space at the time that the binary is relocated. If the binary that's being relocated is an executable, then the process' address space is generally in a pristine state since the executable is one of the first things to be mapped into the address space. As such, the first available address will always be 0x10000 on default installations of Windows. If the binary is a DLL, it's hard to predict what the state of the address space will be in all cases. When a conflict does occur, the kernel searches for an available address region by

²This assumes that the machine that the executable will run on is not running with /3GB. If so, a higher address would have to be used.

traversing from lowest to highest address. For the purposes of this paper, it will be assumed that an executable is being packed and that the address being relocated to is 0x10000. Further research may provide insight into how to better control or alter the expected base address.

With both the preferred base address and the expected base address known, the only thing that remains is to perform the operations that will transform the on-disk version of the binary in a manner that causes custom relocations to restore the binary to its original form at runtime. This process can be both simplistic and complicated. The simplest approach would be to enumerate over the contents of each section in the binary, altering the value at each location by subtracting the displacement and then creating a relocation fixup descriptor that will ensure that the contents are restored to the expected value at runtime. This is how the proof of concept works. A more complicated approach would be to create multiple relocation fixup descriptors per-address. This would mean that the displacement would need to be subtracted once for each fixup descriptor. It should also be possible to apply relocations to individual bytes within a four byte span rather than applying relocations in four byte increments. Even more interesting would be to use some fixup types other than HIGHLOW, although this could be seen as something that might make generating a signature easier.

The end result of this whole process is a functional proof of concept that packs a binary in the manner described above. To get a feel for how different the binary looks after being packed, consider what the implementation of main from earlier in this paper looks like. Notice how the first two instructions are the same as they were previously. This has to do with the fact that base addresses must align on 64KB boundaries, and thus the lower two bottoms are not changed. This could be further improved such as through the strategies described above:

```
.text:84011000 loc_84011000:
.text:84011000   push   ebp
.text:84011001   mov    ebp, esp
.text:84011003   in     al, dx
.text:84011004   add   [eax+0], dh
.text:84011006   add   [edi+edi*8+1209C15h], eax
.text:8401100D   test  [ebx-3FCCFB3Ch], al
.text:84011013   loope near ptr 84010FD8h
.text:84011015
.text:84011015 loc_84011015:
.text:84011015   push  (offset off_8401139C+1)
```

The locate proof of concept has been tested on Windows XP and Windows 2003 Server. Initial testing on Windows Vista indicates that Vista does not properly alter the entry point address after relocations have been applied when an executable is packed. Even though the proof of concept implementation works, there are a number of more fundamental problems with the technique itself.

The first set of problems has to do with techniques that can be used to signature

locreate packed executables. Since locate relies on injecting a large number of relocation fixups, it may be possible to heuristically detect an increased number of relocation fixups with relation to the size of individual segments. This particular attack could be solved by decreasing the number of relocation fixups injected by locate. This would have the effect of only partially mangling the binary, but it might be enough to make people wonder what's going on without giving things away. Even if it weren't possible to heuristically detect an increased number of relocation fixups, it's definitely possible to detect the fact that an executable packed by locate will have an invalid preferred base address that will always result in a conflict. This fact alone makes it mostly trivial to at least detect that something odd is going on.

Detection is only the first problem, however. Once a locate packed executable has been detected, the next logical step is to attempt to figure out some way of obtaining the original executable. Since locate relies on relocation fixups to do this, the only thing one would have to do in order to obtain the original binary would be to relocate the executable to the expected base address that was used when the binary was packed, such as `0x10000`. While it's trivial to develop tools to perform this action, the Interactive Disassembler (IDA) already supports it. When opening an executable, the "Manual Load" checkbox can be toggled. This will cause IDA to prompt the user to enter the base address that the binary should be loaded at. When the base address is entered, IDA processes relocations and presents the relocated binary image. The mitigating factor here is that the user must know the expected base address, otherwise the binary will still appear completely mangled when it's relocated to the wrong base address.

In the author's opinion, these problems make locate a sub-par packer. At best it should be viewed as an interesting approach to the problem of packing executables, but it should not be relied upon as a means of thwarting static analysis. Anyone who reads this paper will have the tools necessary to unpack executables that have been packed by locate. With that said, it should be noted that there is still an opportunity for further research that could help to identify ways of improving locate. For instance, a better understanding of differences in the way the dynamic loader and existing static analysis tools process relocation fixups could provide some opportunity for improvement. Results from some of the author's initial tests of these ideas are included in appendix A. Here's a brief list of some differences that could exist:

1. Different behaviors when processing fixups

It's possible that the dynamic loader and static analysis tools such as IDA may not support the same set of fixup types. Furthermore, they may not process fixup types in the same way. If differences do exist, it may be possible to create a packed executable that will work correctly when used against the dynamic loader but not render properly when relocated using a static analysis tool such as IDA.

2. Relocation blocks with non-page-aligned `VirtualAddress` fields

It's unknown whether or not the dynamic loader and static analysis tools are able to properly handle relocation blocks that have non-page-aligned `VirtualAddress` fields. In all normal circumstances, `VirtualAddress` will be page aligned.

3. Relocation blocks that modify other relocation blocks

An interesting situation that may lead to differences between the dynamic loader and static analysis tools has to do with relocation blocks that modify other relocation blocks. In this way, the relocation information that exists on disk is not what is actually used, in its entirety, when relocating an image during runtime.

Even if research into these topics doesn't yield any direct improvements to locate, it should nonetheless provide some interesting insight into the way that different applications handle relocation processing. And after all, gaining knowledge is what it's really all about.

Appendix A

Differences in Relocation Processing

This appendix attempts to describe some tests that were run on different applications that process relocation entries for binary files. Identifying differences may make it possible to have a binary that will work correctly when executed but not when analyzed by a static analysis tool such as IDA. To test out these ideas, the author threw together a small relocation fuzzing tool that is aptly named `relocfuzz`. This tool will take a pre-existing binary and create a new one with custom relocations. The code for this tool can be found in the other code associated with this paper.

The tests included in this appendix were performed against three different applications: the dynamic loader (`ntdll.dll`), IDA, and `dumpbin`. If the same tests are run against other applications, the author would be interested in knowing the results.

A.1 Non-page-aligned Block `VirtualAddress`

In all normal cases, relocation blocks will be created with a page-aligned `VirtualAddress`. However, it's unclear if non-page-aligned `VirtualAddress`'s will be handled correctly when relocations are processed. There are some interesting implications of non-page-aligned `VirtualAddress`'s. In many applications, such as the dynamic loader, it's critical that addresses referenced through RVAs are validated so as to prevent references being made to external addresses. For example, if relocations were processed in kernel-mode, it would be critical that checks be performed to ensure that RVAs don't end up making it possible to reference kernel-mode addresses. The reason why non-page-aligned `VirtualAddress`'s

are interesting is because they leave open the possibility of this type of attack.

Consider the scenario of a binary that is relocated to `0x7ffe0000`, ignoring for the moment that `SharedUserData` already exists at this location. Now, consider that this binary has a relocation block with a virtual address of `0x1ffff`. This address is not page-aligned. Now, consider that this relocation block has a fixup descriptor that indicates that at offset `0x4` into this page, a certain type of fixup should be performed. This would equate to modifying memory at `0x80000003`, a kernel-mode address. If relocations were being processed in kernel-mode, like they are on Windows Vista for ASLR, then a failure to check that the actual address being written to would result in a dangerous condition.

Here's an example of some code that attempts to test out this idea:

```
static VOID TestNonPageAlignedBlocks(
    __in PPE_IMAGE Image,
    __in PRELOC_FUZZ_CONTEXT FuzzContext)
{
    PRELOCATION_BLOCK_CONTEXT KillerBlock = AllocateRelocationBlockContext(1);

    PrependRelocationBlockContext(
        FuzzContext,
        KillerBlock);

    KillerBlock->Rva = 0x10001;
    KillerBlock->Fixups[0] = (3 << 12) | 0;
}
```

In this example, a custom relocation block is created with one fixup descriptor. The `VirtualAddress` associated with the block is set to `0x10001` and the first fixup descriptor is set to modify offset `0` into that RVA. If the binary that is hosting these relocations is relocated to `0x10000`, a write should occur to `0x20001` when processing the relocations. Here are the results from a few initial tests:

Application	Results
<code>ntdll.dll</code>	The relocation fixup is processed and results in a write to <code>0x20001</code> .
IDA	Ignores the relocation fixup, but only because it writes outside of the executable from what it would appear.
<code>dumpbin.exe</code>	Parses the relocation block without issue.

A.2 Writing to External Addresses

Due to the fact that the `VirtualAddress` associated with each relocation block is a 32-bit RVA, it is possible to create relocation blocks that have RVAs that

actually reside outside of the mapped executable that is being relocated. This is important because if steps aren't taken to detect this scenario, the application processing the relocation fixups might be tricked into writing to memory that is external to the mapped binary. Creating a test-case for this example is trivial:

```
static VOID CreateExternalWriteRelocationBlock(
    __in PPE_IMAGE Image,
    __in PRELOC_FUZZ_CONTEXT FuzzContext)
{
    PRELOCATION_BLOCK_CONTEXT ExtBlock = AllocateRelocationBlockContext(2);

    ExtBlock->Rva = 0x10000;
    ExtBlock->Fixups[0] = (3 << 12) | 0x0;
    ExtBlock->Fixups[1] = (3 << 12) | 0x1;

    PrependRelocationBlockContext(
        FuzzContext,
        ExtBlock);
}
```

In this test, a relocation block is created that has a `VirtualAddress` of `0x10000`. When the binary is relocated to `0x10000`, the actual address of the region that will be written to is `0x20000`. In almost all versions of Windows NT, this address is the location of the process parameters structure. The block itself contains two fixup descriptors, each of which will result in a write to the first few bytes of the process parameters structure. The results after running this test are:

Application	Results
<code>ntdll.dll</code>	The relocation fixup is processed and results in two 32-bit writes to <code>0x20000</code> and <code>0x20001</code> .
IDA	Ignores RVAs outside of the executable.
<code>dumpbin.exe</code>	N/A, <code>dumpbin</code> doesn't actually perform relocation fixups.

A.3 Self-updating Relocation Blocks

One of the more interesting nuisances about the way relocation fixups are processed is that it's actually possible to create a relocation block that will perform fixups against other relocation blocks. This has the effect of making it such that the relocation information that appears on disk is actually different than what is processed when relocation fixups are applied. The basic idea behind this approach is to prepend certain relocation blocks that apply fixups to subsequent relocation blocks. This all works because relocation blocks are typically processed in the order that they appear. An example of this basic concept is described shown below:

```

static VOID PrependSelfUpdatingRelocations(
    __in PPE_IMAGE Image,
    __in PRELOC_FUZZ_CONTEXT FuzzContext)
{
    PRELOCATION_BLOCK_CONTEXT SelfBlock;
    PRELOCATION_BLOCK_CONTEXT RealBlock;
    ULONG RelocBaseRva;
    ULONG NumberOfBlocks = FuzzContext->NumberOfBlocks;
    ULONG Count;

    //
    // Grab the base address that relocations will be loaded at
    //
    RelocBaseRva = FuzzContext->BaseRelocationSection->VirtualAddress;

    //
    // Grab the first block before we start prepending
    //
    RealBlock = FuzzContext->NewRelocationBlocks;

    //
    // Prepend self-updating relocation blocks for each block that exists
    //
    for (Count = 0; Count < NumberOfBlocks; Count++)
    {
        PRELOCATION_BLOCK_CONTEXT RelocationBlock;

        RelocationBlock = AllocateRelocationBlockContext(2);

        PrependRelocationBlockContext(
            FuzzContext,
            RelocationBlock);
    }

    //
    // Walk through each self updating block, fixing up the real blocks to
    // account for the amount of displacement that will be added to their Rva
    // attributes.
    //
    for (SelfBlock = FuzzContext->NewRelocationBlocks, Count = 0;
        Count < NumberOfBlocks;
        Count++, SelfBlock = SelfBlock->Next, RealBlock = RealBlock->Next)
    {
        SelfBlock->Rva = RelocBaseRva + RealBlock->RelocOffset;

        //
        // We'll relocate the two least significant bytes of the real block's RVA
        // and SizeOfBlock.
        //
        SelfBlock->Fixups[0] = (USHORT)((IMAGE_REL_BASED_HIGHLOW << 12) |
            (((RealBlock->RelocOffset - 2) & 0xfff)));
        SelfBlock->Fixups[1] = (USHORT)((IMAGE_REL_BASED_HIGHLOW << 12) |
            (((RealBlock->RelocOffset + 2) & 0xfff)));
        SelfBlock->Rva      &= ~(PAGE_SIZE-1);

        //
        // Account for the amount that will be added by the dynamic loader after

```

```

    // the first self-updating relocation blocks are processed.
    //
    *(PUSHORT)(&RealBlock->Rva)      -= (USHORT)(FuzzContext->Displacement >> 16) + 2;
    *(PUSHORT)(&RealBlock->SizeOfBlock) -= (USHORT)(FuzzContext->Displacement >> 16) + 2;
}
}

```

This test works by prepending a self-updating relocation block for each relocation block that exists in the binary. In this way, if there were two relocations blocks that already existed, two self-updating relocation blocks would be prepended, one for each of the two existing relocation blocks. Following that, the self-updating relocation blocks are populated. Each self-updating relocation block is created with two fixup descriptors. These fixup descriptors are used to apply fixups to the `VirtualAddress` and `SizeOfBlock` attributes of its corresponding existing relocation block. Since a `HIGHLOW` fixup only applies to two most significant bytes, the RVAs of the corresponding fields are adjusted down by two. The end result of this operation is that the first n relocation blocks are responsible for fixing up the `VirtualAddress` and `SizeOfBlock` attributes associated with subsequent relocation blocks. When relocations are processed in a linear fashion, the subsequent relocation blocks are updated in a way that allows them to be processed correctly.

Running this test against the set of test applications produces the following results:

Application	Results
ntdll.dll	The relocation blocks are fixed up accordingly and the application executes as expected.
IDA	Initial testing indicates that IDA is capable of handling self-updating relocation blocks.
dumpbin.exe	Crashes as the result of apparently corrupt relocation blocks: DUMPBIN : fatal error LNK1000: Internal error during DumpBaseRelocations Version 8.00.50727.42 ExceptionCode = C0000005 ExceptionFlags = 00000000 ExceptionAddress = 00443334 NumberParameters = 00000002 ExceptionInformation[0] = 00000000 ExceptionInformation[1] = 7FFA2000 CONTEXT: Eax = 0000000A Esp = 0012E500 Ebx = 00004F00 Ebp = 00000000 Ecx = 7FFA2000 Esi = 00000000 Edx = 781C3B68 Edi = 7FFA2000 Eip = 00443334 EFlags = 00010293 SegCs = 0000001B SegDs = 00000023 SegSs = 00000023 SegEs = 00000023 SegFs = 0000003B SegGs = 00000000 Dr0 = 00000000 Dr3 = 00000000 Dr1 = 00000000 Dr6 = 00000000 Dr2 = 00000000 Dr7 = 00000000

A.4 Integer Overflows in Size Calculations

A potential source of mistakes that could be made when processing relocations has to do with the handling of the `SizeOfBlock` attribute of a relocation block. There is a potential for an integer overflow to occur in applications that don't properly handle situations where the `SizeOfBlock` attribute is less than the size of the base relocation structure (which is 8 bytes). In order to calculate the total number of fixups in a section, it's common to see a calculation like $(\text{Block} \rightarrow \text{SizeOfBlock} - 8) / 2$. However, if a check isn't made to ensure that `SizeOfBlock` is at least 8, an integer overflow will occur. If this happens, the application processing relocations would be tricked into processing a very large number of relocations. An example of a test for this issue is shown below:

```
static VOID TestIntegerOverflow(
```

```

    __in PPE_IMAGE Image,
    __in PRELOC_FUZZ_CONTEXT FuzzContext)
{
    PRELOCATION_BLOCK_CONTEXT EvilBlock = AllocateRelocationBlockContext(0);

    EvilBlock->SizeOfBlock = 0;
    EvilBlock->Rva          = 0x1000;

    PrependRelocationBlockContext(
        FuzzContext,
        EvilBlock);
}

```

In this example, a relocation block is created that has its `SizeOfBlock` attribute set to zero. This is invalid because the minimum size of a block is 8 bytes. The results of this test against different applications are shown below:

Application	Results
ntdll.dll	<p>Does not perform appropriate checks which appears to result in an integer overflow:</p> <pre> (9d4.6dc): Access violation - code c0000005 (first chance) First chance exceptions are reported before any exception handling. This exception may be expected and handled. eax=00000000 ebx=00014008 ecx=00011000 edx=80010000 esi=00015000 edi=ffffffff eip=7c91e163 esp=0013fa98 ebp=0013faac iopl=0 nv up ei pl nz na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206 ntdll!LdrProcessRelocationBlockLongLong+0x1a: 7c91e163 0fb706 movzx eax,word ptr [esi] ds:0023:00015000=???? </pre>
IDA	<p>Ignores the relocation block, but may not process relocations correctly as a result (unclear at this point).</p>
dumpbin.exe	<p>Refuses to show relocations:</p> <pre> Microsoft (R) COFF/PE Dumper Version 8.00.50727.42 Copyright (C) Microsoft Corporation. All rights reserved. Dump of file foo.exe File Type: EXECUTABLE IMAGE BASE RELOCATIONS #4 Summary 1000 .data 1000 .rdata 1000 .reloc 1000 .text </pre>

A.5 Consistent Handling of Fixup Types

Applications that process relocation fixups may also differ in their level of support for different types of fixups. While most binaries today use the HIGHLOW fixup exclusively, there are still quite a few other types of fixups that can be applied. If differences in the way relocation fixups are processed can be identified, it may be possible to create a binary that relocates correctly in one application but not in another application. The following code demonstrates an example of this type of test:

```
static VOID TestConsistentRelocations(
    __in PPE_IMAGE Image,
    __in PRELOC_FUZZ_CONTEXT FuzzContext)
{
    PRELOCATION_BLOCK_CONTEXT Block = AllocateRelocationBlockContext(16);
    ULONG Rva = FuzzContext->BaseRelocationSection->VirtualAddress;
    INT Index;

    PrependRelocationBlockContext(
        FuzzContext,
        Block);

    Block->Rva = 0x1000;

    for (Index = 0; Index < 16; Index++)
    {
        //
        // Skip invalid fixup types
        //
        if ((Index >= 6 && Index <= 8) ||
            (Index >= 0xb && Index <= 0x10))
            continue;

        Block->Fixups[Index] = (Index << 12) | Index;
    }
}
```

This test works by prepending a relocation block that contains a relocation fixup for each different valid fixup type. This results in a relocation block that looks something like this:

```
BASE RELOCATIONS #4
1000 RVA,      28 SizeOfBlock
 0 ABS
 1 HIGH          EC8B
 2 LOW          8BEC
 3 HIGHLOW      5008458B
 4 HIGHADJ      0845 (5005)
 0 ABS
 0 ABS
 0 ABS
 9 IMM64
```

```

A DIR64      8000209C15FF8000
0 ABS
0 ABS
0 ABS
0 ABS
0 ABS
0 ABS

```

The results for this test are shown below:

Application	Results
ntdll.dll	<p>While not confirmed, it is assumed that the dynamic loader performs all fixup types correctly. This results in the following code being produced in the test binary:</p> <pre> foo+0x1000: 00011000 55 push ebp 00011001 8c6c8b46 mov word ptr [ebx+ecx*4+46h],gs 00011005 895068 mov dword ptr [eax+68h],edx 00011008 1830 sbb byte ptr [eax],dh 0001100a 0100 add dword ptr [eax],eax 0001100c 00b69b200100 add byte ptr foo+0x209b (0001209b)[esi],dh 00011012 83c408 add esp,8 </pre>
IDA	<p>Appears to handle some relocation fixup types differently than the dynamic loader. The result of IDA relocating the same binary results in the following being produced:</p> <pre> .text:00011000 push ebp .text:00011001 mov ebp, esp .text:00011003 mov eax, [ebp+9] .text:00011006 shr byte ptr [eax+18h], 1 ; "Called TestFunction()\n" .text:00011009 xor [ecx], al .text:00011009 .text:0001100B db 0 .text:0001100C .text:0001100C add byte ptr ds:printf[esi], dl .text:00011012 add esp, 8 </pre> <p>Equates to:</p> <pre> .text:00011000 55 8B EC 8B 45 09 D0 68 18 30 01 00 00 96 9C 20 .text:00011010 01 00 83 C4 08 C7 05 50 </pre>
dumpbin.exe	N/A, dumpbin doesn't actually perform relocation fixups.

A.6 Hijacking the Dynamic Loader

Since the dynamic loader in previous tests proved to be capable of writing to areas of memory external to the executable binary, it makes sense to test to see if it's possible to hijack execution control. One method of approaching this would be to have the dynamic loader apply a relocation to the return address

of the function used to process relocations. When the function returns, it'll transfer control to whatever address the relocations have caused it to point to. An example of this code for this test is shown below:

```
static VOID TestHijackLoader(
    __in PPE_IMAGE Image,
    __in PRELOC_FUZZ_CONTEXT FuzzContext)
{
    PRELOCATION_BLOCK_CONTEXT Block = AllocateRelocationBlockContext(1);

    PrependRelocationBlockContext(
        FuzzContext,
        Block);

    //
    // Set the RVA to the address of the return address on the stack taking into
    // account the displacement.
    //
    Block->Rva      = 0x0012fab0;
    Block->Fixups[0] = (3 << 12) | 0;
}
```

When a binary is executed that contains this relocation block, the dynamic loader ends up applying a relocation to the return address located at 0x13fab0. Obviously, this address may be subject to change quite frequently, but as a means of illustrating a proof of concept it should be sufficient. And, just as one would expect, the dynamic loader does indeed overwrite the return address and make it possible to gain control of execution:

```
(c88.184): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0001400a ebx=00014008 ecx=0013fab0 edx=80010000 esi=00000001
edi=ffffffff eip=fc92e10b esp=0013fac8 ebp=0013fae4 iopl=0   nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000  efl=00010246
fc92e10b ??             ???
0:000> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0013fac4 00010000 00261f18 7ffdc000 80010000 0xfc92e10b
0013fae4 7c91e08c 00010000 00000000 00000000 image00010000
0013fb08 7c93ecd3 00010000 7c93f584 00000000 ntdll!LdrRelocateImage+0x1d (FP0: [Non-Fpo])
0013fc94 7c921639 0013fd30 7c900000 0013fce0 ntdll!LdrpInitializeProcess+0xea0 (FP0: [Non-Fpo])
0013fd1c 7c90eac7 0013fd30 7c900000 00000000 ntdll!_LdrpInitialize+0x183 (FP0: [Non-Fpo])
00000000 00000000 00000000 00000000 00000000 ntdll!KiUserApcDispatcher+0x7
```

Bibliography

- [1] Carrera, Ero. *Packer Tracing*.
<http://nzight.blogspot.com/2006/06/packer-tracing.html>; accessed Dec 15, 2006.
- [2] Szor, Peter. *Advanced Code Evolution Techniques and Computer Virus Generator Kits*.
<http://www.informit.com/articles/article.asp?p=366890&seqNum=3&rl=1>; accessed Jan 8, 2007.
- [3] Szor, Peter. *Tricky Relocations*.
<http://peterszor.com/resurrel.pdf>; accessed Jan 11, 2007.