

# Anti-RE Techniques in DRM Code

Jan Newger  
Seminar Advanced Exploitation Techniques  
RWTH Aachen, Germany  
jan.newger@rwth-aachen.de

## ABSTRACT

In order to prevent music from being copied among consumers, content providers often use DRM systems to protect their music files. This document describes the approach taken while analysing a DRM system (whose identity needs to be kept secret due to legal issues)<sup>1</sup>. It is shown what techniques were used to protect the system from being easily reverse engineered.

## 1. INTRODUCTION

It's common practice among DRM implementations to use strong encryption in combination with a hardware or user dependent key derivation algorithm. The DRM system in question is no exception to this, although the algorithm used for encryption, single DES [1] in this case, is considered to be outdated because of its limited key length. Because the whole system can be broken by revealing the decryption algorithm together with its associated key setup, the central idea of the protections used by the DRM is to make reverse engineering as hard as possible. Although any software based protection mechanism can be reverse engineered and therefore be broken, performing such a task can be highly hindered by using various anti reverse engineering techniques. This paper will give a detailed analysis of the most significant techniques employed by the DRM and shows how to circumvent them.

## 2. APPROACHING THE DRM

The most important step while approaching the protection was to find DRM related code which would finally lead to the decryption algorithm.

A very straight forward strategy to solve this problem is to use a debugger to find relevant code by setting breakpoints on file I/O APIs like `CreateFile`, `ReadFile` and functions used to map file data in process memory by means of memory mapped files. After data has been read from a DRM

<sup>1</sup>Please note that this paper lacks quite some details, because i don't want to offend the developers of the DRM

protected file, one would set a break point on memory access (BPM), which in turn would lead to either a copy operation or directly to the code decrypting the file buffer. Under the assumption that the code contributing to the key setup is rather close to the decryption algorithm, this strategy seems to be quite appropriate in this scenario. But even if that was not the case, it would still be possible to back trace from the code which accesses the key to the key schedule algorithm itself, e.g. also by using BPMs. The designers of the DRM system were obviously expecting this or a similar approach to be used by a potential attacker, so they came up with a protection which makes it impossible to use BPMs without further action by the attacker. Chapter 3 discusses techniques to reclaim the features offered by the hardware breakpoints, so BPMs can be used as proposed.

### 2.1 Code Coverage

Another approach which can be quite effective when it comes to finding relevant code in large binaries is to use code coverage. In this context code coverage is the process of identifying basic blocks or functions inside a binary which have been executed during runtime. For the purpose of finding DRM relevant code, a tool, namely N-Coverage<sup>2</sup>, has been developed. The application consists of a plugin for the Interactive Disassembler (IDA) [2] and a stand-alone application written in C#. The plugins purpose is to export relative virtual addresses (RVA) for each function or basic block gathered from the disassembly of a library or executable. This information is fetched from IDA and can then be exported to the stand-alone application, which in turn creates a new process and attaches a custom debugging engine to set breakpoints in the specified modules and/or executable images. While the process is running, breakpoint hits are recorded and are saved on a per-module basis. So as to handle large applications consisting of many DLLs with possibly conflicting image base addresses, N-Coverage is able to correctly handle library rebasing. In a final step N-Coverage allows the user to merge and diff recordings resulting in a new set which can be exported back to the IDA plugin again. This allows for easy visualization of functions or basic blocks contained in the final set, either by coloring the respective disassembly listing or by selecting hits from a list so they can be easily navigated to. The problem of finding DRM relevant code can be accomplished by recording a set of hits while playing non-DRM protected music first, saving the hits to a set *s1*. Afterwards another set of hits *s2* is recorded, but this time while playing DRM protected music. DRM specific code

<sup>2</sup>Available from [www.newgre.net/ncoverage](http://www.newgre.net/ncoverage)

can then be found by computing the final set  $s = \{s2 \setminus s1\}$ . This works best if all or most of the hits for irrelevant code, like e.g. GUI related code, common initialization routines and so on, have been recorded to  $s1$ , so these hits will be filtered out of  $s2$ , leaving only relevant hits in the final set  $fs$ . Against this background code coverage seems to be a promising strategy to find code contributing to the DRM, especially because it doesn't require any analysis to be carried out by the reverse engineer. It turns out however, that in this case code coverage is only of limited use. This is due to runtime code modifications executed by the DRM which makes the process of relating breakpoint hits to RVAs very hard, because previously assumed break point addresses are never hit due to code which copies itself to other locations in memory. Although code coverage gave a few good starting points, it wasn't appropriate to be used as the key strategy in this case. Therefore the basic approach was based on using BPMs to locate code of interest.

## 2.2 Introduction to Windows SEH

The major aspects of the anti reverse engineering techniques used in the DRM system rely on the mechanisms of structured exception handling (SEH), so a short overview on the architecture of exception handling on operating system level under windows is given. This is only a rough overview of structured exception handling, so more advanced topics such as stack unwinding, nested exception etc. are intentionally left out. A more detailed and complete discussion of the topic can be found at [3].

Structured exception handling is provided by the operating system to allow an application to react on runtime errors on a per-thread basis. An exception handler called by means of SEH has the following signature:

```
EXCEPTION_DISPOSITION _except_handler(
    _EXCEPTION_RECORD* ExceptionRecord,
    void* EstablisherFrame,
    _CONTEXT* ContextRecord,
    void* DispatcherContext
);
```

Listing 1: Handler declaration

The most important parameters for our analysis are `EXCEPTION_RECORD` and `CONTEXT`. The former parameter contains information like the exception code, the address where the exception occurred, etc. whereas the latter is a pointer to a structure representing the CPU state at the time of the exception, i.e. the thread context of the faulting thread. The supplied context actually is an out-parameter, so any changes made to it will be applied by the operating system upon return of the handler. This allows a certain handler to fix whatever caused the exception in the first place by modifying the given thread context. Having one global exception handler which is responsible for processing any possible error in a given thread is often unsuitable, so there is a linked list of `EXCEPTION_REGISTRATION` structures pointed to by `fs:0`. This allows for registration of multiple exception handlers per thread, especially for different scopes. Listing 2 shows the entries of this linked list.

```
_EXCEPTION_REGISTRATION struc
prev    dd    ?
```

```
handler dd    ?
_EXCEPTION_REGISTRATION ends
```

Listing 2: SEH list entry

This is a very straight forward way of implementing a linked list, `prev` is a pointer to the previous element in the list (or `0xFFFFFFFF` to mark the last element) and `handler` obviously is a pointer to the respective exception handler. Whenever an exception occurs, the operating system walks the list of `EXCEPTION_REGISTRATION` structures of the respective thread starting at `fs:0` and calls each handler until the first replies to handle the exception, signaling the operating system that this handler is capable of fixing whatever caused the exception in the first place.

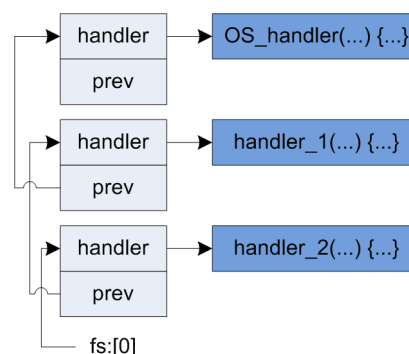


Figure 1: SEH linked list

The SEH list contains at least one entry which is inserted by the operating system executable loader and jumps in to catch any unhandled exception, leading usually to process termination. Each thread can register a new exception handler by inserting a new `EXCEPTION_REGISTRATION` structure to the beginning of the SEH list. Listing 3 shows code to add a new exception handler. Although this is probably the most basic code to do this, nevertheless in most applications the compiler generated code to add a new handler looks very similar to this.

```
push    handler
push    fs:[0]
mov     fs:[0], esp
```

Listing 3: Adding a new handler

A new `EXCEPTION_REGISTRATION` structure is created on the stack. Since the stack grows from higher to lower addresses, the second member of the structure has to be pushed first. The `prev` pointer is the last pointer from the list, i.e. `fs:[0]`. Finally the pointer to the structure on the stack is saved as the new head of the list. In most cases one exception handler is responsible to process all exceptions in a special scope, so before entering this certain scope, an exception handler is added in the way just shown. Against this background it makes sense to create the `EXCEPTION_REGISTRATION` structure on the stack, because as soon as control flow leaves the scope protected by the handler, it can be safely removed from the stack.

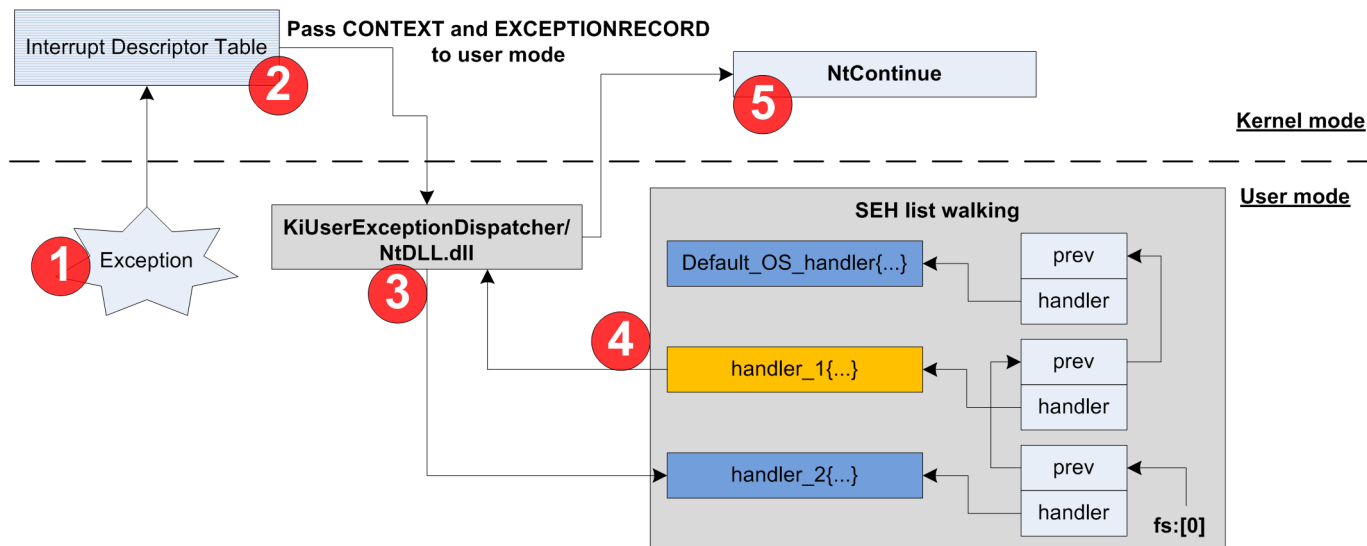


Figure 2: SEH big picture

```

mov    eax, [esp]
mov    fs:[0], eax
add    esp, 8

```

Listing 4: Removing a handler

To unregister a handler, the previous element is set as the new head element and the `EXCEPTION_REGISTRATION` structure is removed from the stack as shown in listing 4.

All the SEH logic is initially triggered by code from a procedure namely `KiUserExceptionDispatcher` exported by `ntdll.dll` (though it's not really a procedure, but rather code being dispatched to from kernel mode). This means that whenever an exception is raised, the CPU transfers control flow into kernel mode and the appropriate interrupt service routine from the interrupt descriptor table is grabbed and executed. In kernel mode some information about the exception is collected as well as the context of the faulting thread and finally the structures containing this information are passed down to user mode ending up in `KiUserExceptionDispatcher`. It turns out that `KiUserExceptionDispatcher` is indeed the first code being executed in user mode after the exception has occurred.

```

KiUserExceptionDispatcher (PEXCEPTION_RECORD
    pExcptRec,
    CONTEXT* pCtx)
{
    DWORD retValue;
    if (RtlDispatchException(pRec, pCtx))
        retValue = NtContinue(pContext, 0);
    else
        retValue = NtRaiseException(pRec, pCtx, 0);

    EXCEPTION_RECORD rec;
    rec.ExceptionCode = retValue;
    rec.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
    rec.ExceptionRecord = pExcptRec;
    rec.NumberParameters = 0;
}

```

```

    RtlRaiseException(&rec);
}

```

Listing 5: `KiUserExceptionDispatcher` pseudo code

All of the SEH logic in turn is triggered from `RtlDispatchException`. This procedure creates the parameters an exception handler expects and then walks the list of registered exception handlers. If no handler was found during SEH list walking, which responded to handler the exception, a second chance exception is raised by means of `NtRaiseException`, which leads to process termination<sup>3</sup>. If one of the two system calls returns to `KiUserExceptionDispatcher` some serious bug appeared and an exception is raised by means of `RtlRaiseException`. If on the other hand a suitable handler has been found, the handler has two choices. It can

- return so control flow again resumes in `KiUserExceptionDispatcher`
- decide not to return, which has the effect, that control flow will neither come back to `RtlDispatchException` nor to `KiUserExceptionDispatcher`

All the handlers participating in the protection are of the former type, whereas non-returning handlers are common among exception handling code generated by C++ compilers<sup>4</sup>. So if the handler actually returns, a possibly modified context is applied by means of `NtContinue` and the faulting thread is resumed on the next schedule.

### 3. ANTI REVERSE ENGINEERING TECHNIQUES

<sup>3</sup>Though an attached debugger gets the opportunity to fix this *second chance exception* before the process is terminated

<sup>4</sup>At least MS compilers behave that way

Anti Reverse engineering techniques can be used to achieve different kinds of effects, which can make the analysis of an executable very hard. Several anti debugging techniques, aiming at online analysis by using a debugger, have been developed[4]. Most of these techniques are highly operating system specific and utilize the fact, that the state of a process being debugged is distinguishable from an untouched process. This is due to the fact that the operating system and the application behave differently in some situations, e.g. the operating system needs to keep additional information in a process to mark it as being debugged (PEB, Debugheap, etc). In addition to that it is also possible to make static reverse engineering a difficult process. Code obfuscation techniques like injection of junk code, code transformations or even mechanisms including fully fledged virtual machines have been developed. The DRM makes use of techniques against both static and dynamic reverse engineering for the sake of complicating the analysis of the DRM system. The key protections include techniques such as trampolines to obfuscate control flow, occupying the debug registers and using them to alter control flow, runtime checks of critical APIs for breakpoint opcodes, heavy use of exceptions to interrupt flow of execution and finally a P-Code machine which encapsulates the decryption and key setup algorithms. The following paragraphs examine these techniques more closely and will also introduce concepts on how to circumvent them or at least show how to ease their impact.

### 3.1 Trampolines

The first technique one comes across while analysing the DRM protection is the use of a mechanism, which will be denoted throughout the paper as *trampolines*<sup>5</sup>. The protection system allocates a few mega bytes of memory on the heap at startup and uses this memory later on to store code and execute it from there. The trampolines serve as a starting point for all other anti reverse engineering techniques, i.e. whenever a new file buffer of DRM protected data needs to be decrypted, flow of execution starts at a central procedure inside the protection. This procedure then prepares some internal data structures needed for managing trampoline state and sets up structures for memory management of the P-Code machine. After initialization the *BeingDebugged* flag in the PEB<sup>6</sup> is checked,

```
mov    eax, large fs:18h
mov    eax, [eax+30h]
movzx  eax, byte ptr [eax+2]
```

Listing 6: Basic debugger check

whereas `fs:18h` is the linear address of the TEB<sup>7</sup> for the executing thread. At offset `30h` is the pointer to the PEB which holds the *BeingDebugged* flag. The PEB entry can be trivially patched with zero in order to fool the detection, because it just marks the process as being debugged but has no further meaning regarding debugging functionality. This flag has been well known for years and is therefore considered

<sup>5</sup>The term *trampoline* was borrowed from the area of shell coding

<sup>6</sup>Process environment block

<sup>7</sup>Thread environment block

to be a very weak debugger detection technique. The second check tries to detect an attached debugger by issuing a fake breakpoint exception. A debugger can be easily hide from this check by just passing the resulting exception back to the process, so at runtime the code behaves in the same way as if it was not running under a debugger. If no debugger has been found by these checks, the procedure in question sets the thread affinity of the current thread, forcing it to run on a randomized CPU in the system<sup>8</sup>. Before control flow is handed over to the first trampoline, the current thread context is fetched by means of the `GetThreadContext` API in order to modify the debug registers, which are used to pass parameters between trampolines and also serve as a storage mechanism to hold the address of the starting trampoline. Finally the modified context is applied by using the `SetThreadContext` API and control flow is transferred to the first trampoline.

#### 3.1.1 Trampoline control flow

Control flow between trampolines isn't dispatched in a standard way with instructions like `call` or `jmp`, as in the case of compiler generated code. Instead control flow heavily relies on exception handling and an internal call stack, which is maintained by the system, so a call hierarchy can be realized between trampolines. Figure 3 shows a situation where the flow of execution starts at `trampolineA` and is supposed to end up at `trampolineB`. Whenever a trampoline initiates such a change of control flow, this process always starts at `trampoline0`. This trampoline is also the first one which is called from the aforementioned procedure in the protection. The address of this trampoline is randomized at runtime via the `RDTSC` instruction (indicated by overlapping semi-transparent boxes). The major tasks this trampoline performs are to copy the next trampoline (`trampoline1`) to a random location and to put the destination trampoline (`trampolineB` in this case) on the internal call stack. This internal call stack is needed to realize a call hierarchy between trampolines, because there is never a direct `call` instruction between trampolines but control flow depends on jumps and exceptions. As a consequence there is no mechanism which implicitly puts a return address on the stack to let control flow return from a nested call, so all of this logic has to be emulated by the protection. `trampoline1` is accountable for copying the previous trampoline to a random location, installing a new exception handler and for raising a single step exception by means of code shown in listing 7. Moreover it copies parameters to a private stack area, which is used by the exception handler to forward them to the next trampoline.

```
pushf
pop    eax
or     eax, 100h
push  eax
popf
```

Listing 7: Raise single step exception

First the `EFLAGS` register is pushed on the stack, the `TF` bit is enabled, and the modified `EFLAGS` register is applied again, so before the next instruction executes, a debug exception is

<sup>8</sup>Reasoning behind this remains unclear at this time

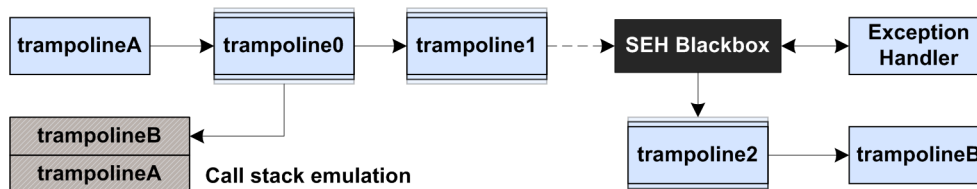


Figure 3: Control flow between trampolines

generated, which ultimately ends up in the previously registered exception handler. This exception handler then alters control flow by changing the instruction pointer based on the parameters copied by the previous trampoline. The handler clears the TF bit<sup>9</sup>, removes the SEH entry from the handler list and gives control back to the operating system. As soon as the thread is scheduled for execution, control flow resumes at `trampoline2` which copies the destination trampoline and finally jumps to `trampolineB`.

Because of the fact that there is no classical call hierarchy between trampolines, a mechanism must exist which allows the system to perform a *return operation*, i.e. whenever a trampoline has finished its operations control flow must resume in the trampoline which invoked the respective trampoline. Once a trampoline wants to leave its scope, it registers a special exception handler and again raises a single step exception. The handler is then called by means of SEH as usual, removes the SEH entry from the handler list, cleans the stack and sets the EIP register to the value found in the DR2 register of the supplied context. After the operating system has applied the modified context, execution resumes at a trampoline whose position is again randomized. This trampoline finally removes the returning trampoline from the internal call stack, copies the code where control flow should resume and returns to this location.

While dispatching control flow to trampolines and back, the DRM system modifies the debug registers by means of the `SetThreadContext` API. Hardware breakpoints are switched on and off repeatedly by modifying the DR7 register to interfere with a possibly attached debugger. The debug registers are used by the trampolines in the following way:

- DR0 and DR6 are mostly zeroed out and don't serve a special purpose
- DR1 contains a pointer to a shared stack area which is used to pass data between trampolines
- DR2 holds the address of the trampoline, which is used to return from another trampoline
- DR3 holds the address of the starting trampoline (`trampoline0`). The address is obfuscated by XORing it with `0x7FFFFFFF`
- DR7 is used to turn hardware breakpoints on and off very frequently

The debug registers DR0 to DR3 are normally used to specify the linear address of a hardware breakpoint, while DR6

<sup>9</sup>A single step exception is a *trap*, so it's not necessary to clear the TF bit to let the program continue normally[5]

and DR7 control options and breakpoint conditions. So by overwriting the debug registers, the breakpoint mechanism becomes unavailable for any attached debugger.

### 3.1.2 Impact

The main purpose of using the trampoline mechanism was probably to make the problem of finding DRM relevant code more difficult. Since control flow between trampolines isn't dispatched in a standard way a disassembler can't easily obtain any cross referencing information, which makes it rather difficult to analyze the dependencies between different trampolines. Additionally, without understanding the mechanisms used to emulate the return logic, it is also difficult to examine the call hierarchy at runtime because it is not possible to perform an *execute until return* operation which is supported by most debuggers. On the other hand as soon as this mechanism is understood, one gets a perfect call stack by watching the internal call stack emulation structure. This is obviously an advantage compared to the standard case where a perfect call stack is not available in general. Since most trampolines don't even have a `ret` instruction, deducing function boundaries also becomes harder for disassemblers. A further effect of the trampolines is the jittering of start addresses caused by the `RDTSC` instruction. This obviously only affects debugging and makes it a rather annoying process, because the disassembler gets confused by changing function boundaries overlapping at the same address. This impact can be alleviated as will be shown in the next section.

The most severe impact is in fact caused by the usage of the debug registers, because this technique effectively blocks all hardware breakpoints. So the strategy of using BPs to watch access attempts on the file buffer becomes infeasible.

### 3.1.3 Ease Impact of Trampoline Randomization

The result of the `RDTSC` instruction is used as the seed for a PRNG, so the jittering of the trampolines can be defeated by changing the instruction result to a constant value. As a consequence, this fixes the trampolines at a constant address, which makes it easier to debug and understand the code. Fortunately, the X86 CPU allows us to turn `RDTSC` into a privileged instruction by modifying the TSD flag of the CR4 register. This implicates that whenever `RDTSC` is executed from a privilege level other than ring0, a general protection exception (`#GP`) is thrown. This exception is classified as a *fault*, which means that the state of the program is saved by the processor prior to the beginning of execution of the faulting instruction. So by writing a driver to patch the interrupt descriptor table (IDT), it is possible to insert a handler, which intercepts this exception event and changes the return value accordingly. As shown in figure 4 an error code along with the instruction pointer and some other reg-

isters are then passed to the exception handler, i.e. the far pointer of the descriptor at offset 13 in the IDT.

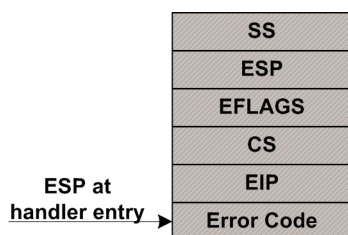


Figure 4: Stack layout of #GP handler

The replaced handler has to make sure that the exception originated in user-mode by checking that  $EIP < 0x80000000$ . Moreover it has to verify that the instruction which caused this fault actually was of type RDTSC. This can be achieved by disassembling the code at EIP. If these preconditions are met, the handler can set  $EDX:EAX$  to a constant value, clean the stack and return from the handler by means of IRETD. In all other cases the handler just cleans its private stack and branches to the original handler.

When loaded, this driver causes each trampoline to be always copied to the same address, which makes debugging a little bit more comfortable. Especially in the phase of analyzing the whole trampoline mechanism, recognizing trampolines becomes a lot easier when using the driver because of the fixed locations.

A technique to completely disable the trampoline mechanisms could have been to set the destination trampoline address to the source address, so trampolines would be in fact not copied at all, but execute from their original source location. This goal could be reached by modifying the trampoline control structures used internally as well as the respective exception handlers, though this approach would have required a considerable amount of work in the first place.

## 3.2 Unblocking the Debug Registers

As previously described the debug registers are used to pass parameters between trampolines and are also used to alter control flow from exception handlers. By using the registers in such a way, program logic heavily depends on the values stored in these registers. This means that it is impossible to just patch out all code related to modification of the debug registers. Instead the context APIs need to be emulated and the central exception dispatcher of the operating system (`KiUserExceptionDispatcher`) has to be modified for the debug registers to be available for debugging purposes. A proven mechanism to hook such API functions is to first inject a DLL into the respective process and then perform inline patching in order to dispatch control flow to an internal hook function.

### 3.2.1 DLL Injection and API Hooking

There are numerous ways of injecting a DLL into a process under windows like using `SetWindowsHookEx`, shellcode injection or the method of using `CreateRemoteThread`[6]. Since this topic has been widely discussed over the past years only the basic ideas are presented. A very reliable and flexible, though platform dependent, method is to inject some shellcode into the target process. This shellcode can then

load the DLL in question from inside the target process. First of all the control process allocates some memory in the target process by using the `VirtualAllocEx` API function. Memory has to be allocated for the shellcode as well as for a data structure which is used to pass the DLL path to the shellcode and to read back error codes. In the second step, the control process injects the actual shellcode via the `WriteProcessMemory` API and then creates a new thread at that address by means of `CreateRemoteThread`. The shellcode in turn loads the DLL, saves the DLL handle to the previously allocated data structure and terminates itself. The injecting process waits until the thread handle gets signaled and reads back the DLL handle or an error code by means of `ReadProcessMemory`. This handle can then be used to perform remote calls at will in the target process. Figure 5 illustrates this approach. Once inside the address space of the target process, the DLL can hook into any API function used by the target process, so it is capable of modifying any functionality exposed by imported API functions. Since the DRM system builds custom stubs which scan API functions for `int3` opcodes and directly jump to the respective API, inline patching is an adequate solution to this problem. API hooking was realized by using the detours package[7] available from Microsoft Research.

### 3.2.2 Context Emulation

The first step in preventing the DRM system from modifying the debug registers is to hook the `SetThreadContext` and `GetThreadContext` APIs, so upon invocation a hook function is executed and redirects set and get requests to an internal storage. This obviously makes it impossible for the protection system to modify the debug registers, which allows a debugger to use them. The problem with this approach is, that as soon as control flow reaches an exception handler, which emulates the return logic of a trampoline, the supplied context is out of sync with the context saved in the internal storage of the injected DLL. The reason for this is, that the operating system itself passes the *real* thread context of the faulting thread from kernel mode down to `KiUserExceptionDispatcher`. From there, the thread context is forwarded to `RtlDispatchException` and finally ends up in the respective exception handler. By placing an additional hook in `KiUserExceptionDispatcher` it is possible to re-synchronize the two contexts again, so the DRM system gets the expected values passed to the exception handler on the one hand, and the debugger can use the debug registers to place hardware breakpoints on the other hand.

```

mov     ecx, [esp+4]
mov     ebx, [esp+0]
push   ecx
push   ebx
call   RtlDispatchException
or     al, al
jz     short loc_7C91EB0A
pop    ebx
pop    ecx
push   0
push   ecx
call   ZwContinue
jmp    short loc_7C91EB15

```

loc\_7C91EB0A:

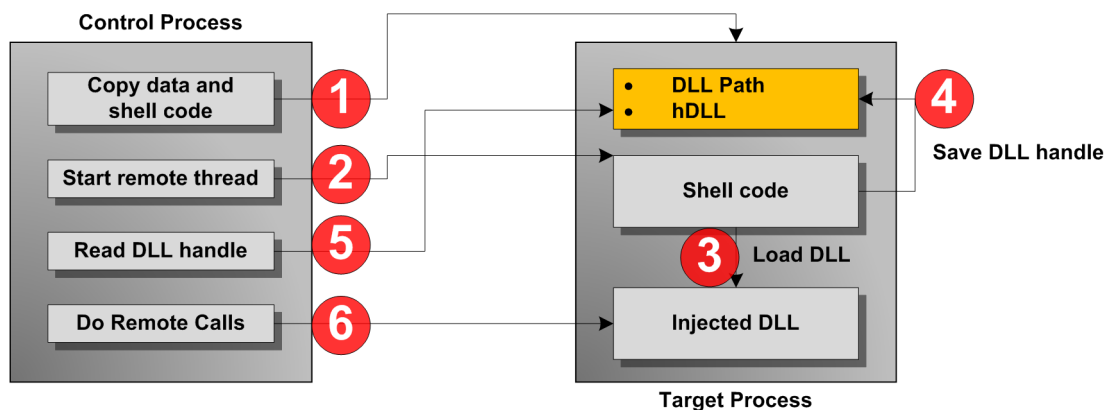


Figure 5: DLL Injection

```

pop     ebx
pop     ecx
push   0
push   ecx
push   ebx
call   ZwRaiseException

loc_7C91EB15:
add     esp, 0FFFFFFECh
mov     [esp], eax
mov     dword ptr [esp+4], 1
mov     [esp+8], ebx
mov     dword ptr [esp+10h], 0
push   esp
call   RtlRaiseException
retn   8

```

Listing 8: KiUserExceptionDispatcher

Listing 8 shows the disassembly of `KiUserExceptionDispatcher`. By looking at the disassembly it becomes obvious why this procedure is named *dispatcher* and that's because there is no return address on the stack so the first parameter is accessible through `[esp+0]` rather than through `[esp+4]` as is the case with normal procedures. Care must be taken when transferring control flow from the hooked `KiUserExceptionDispatcher` to kernel mode, otherwise random blue screens will occur due to a messed up stack<sup>10</sup>. The two parameters pushed to `RtlDispatchException` are the thread context (ECX) and the `EXCEPTION_RECORD` (EBX). `RtlDispatchException` in turn is responsible for all the SEH logic previously discussed. In the context of the DRM system this function will always return, signaling that an appropriate handler has been found during handler list traversal. This means that the call to `ZwContinue` will always be executed. This is the system call to apply a possibly modified context, which also means that this function won't ever return unless there is a severe error. In that case an exception is thrown by means of `RtlRaiseException`. In order to fool the DRM one needs to pass a manipulated context to the exception handler, so the program logic of the DRM system works with the expected values previously set by the `SetThreadContext` API. As soon as `RtlDispatchException`

<sup>10</sup>Though the exact circumstances haven't been investigated

returns, the debug registers in the modified context must be replaced with the values from the real context which came from the operating system kernel. This is important because these values might be in use by a possibly attached debugger. One possible strategy to solve this problem is to re-implement `KiUserExceptionDispatcher`. This is complicated a little by the fact that `RtlDispatchException` is not exported by `ntdll.dll`, so this function must be re-implemented as well. Fortunately this function doesn't need to reassemble all the logic found in `RtlDispatchException`. In case of a single step exception the first handler always terminates SEH list walking, so the hook function only has to prepare all parameters for the exception handler and call it. Listing 9 shows the prologue of the hooked `KiUserExceptionDispatcher`. Because of the fact that this is not a standard procedure, an additional element has to be pushed onto the stack, so the compiler generates valid code to access function parameters.

```

xor     eax, eax
push   eax
push   ebp
mov     ebp, esp
sub     esp, __LOCAL_SIZE

```

Listing 9: Prologue of the hook function

The compiler variable `__LOCAL_SIZE` gives the needed stack space allocated by compiler generated code. This value is needed because the hook function is declared with `__declspec(naked)`, which means that the function prologue and epilogue have to be manually crafted. After the prologue the hook function has to check if the exception is of type single step and if this is the case, the corresponding values for the debug registers are retrieved from the internal storage based on the current thread ID. All parameters the handler expects are then prepared and are passed along with the manipulated context to the first exception handler found at `[fs:0]`. The handler in turn modifies the context, so upon return all modified parts have to be merged with the real context supplied by the operating system. Afterwards a call is made to `NtContinue` similar to the original implementation found in `KiUserExceptionDispatcher`. Because control flow at this point is in the hooked function and the call to

`NtContinue` won't return, care needs to be taken concerning stack cleanup. The stack must be reset to the state as if the hooked function would have never executed. This is shown in listing 10.

```

mov     ecx, pContext
mov     edx, [NtContinue]
add     esp, __LOCAL_SIZE
pop     ebp
pop     eax
xor     eax, eax
push   eax
push   ecx
call   edx

```

**Listing 10: Applying the manipulated context**

First of all the modified context is fetched and the stack space used by the compiler generated code is cleaned up. The next step is to reset the original base pointer and pop the fake return address from the stack. Finally `NtContinue` with a pointer to the manipulated context is called and the current thread will be resumed with the new context on the next schedule. If the exception is not of type single step the original implementation of `KiUserExceptionDispatcher` is called in a similar way.

By using the techniques outlined in this section, an attached debugger is able to provide the features of hardware breakpoints, so the strategy as proposed in the beginning can be carried out.

### 3.3 P-Code Machine

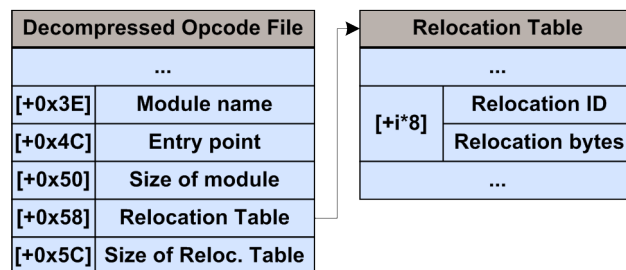
The last obstacle which needs to be taken is to analyze the usage of a P-Code machine which is used to implement the actual decryption algorithm and the associated key setup. The P-Code machine is stack based, so all parameters to the opcodes are pushed and popped of the machine stack. Besides this implementation includes a special register, which receives the result of the respective operation. The instruction set of the emulated CPU overlaps to some extent with the instruction set of the X86 architecture, especially with regard to the arithmetic instructions. Apart from that, the arithmetic instructions of the emulated CPU aren't involved in the decryption or in the key setup, so these haven't been analyzed thoroughly. All in all, the CPU has 256 different opcodes with a fixed length of 1 byte. The set of available opcodes is split into two parts. The first part contains opcodes which are responsible for emulating operations of the CPU itself, like stack manipulation, arithmetic instructions and so on. In contrast, the other opcodes are used to dispatch control flow to handlers containing native code for high level operations, e.g. there are handlers to load opcode modules or to allocate memory from the host machine. The algorithms for decryption of the DRM protected content as well as the routines involved in the key setup are also triggered by means of these high level opcodes.

#### 3.3.1 Opcode Module Files

The actual P-Code is not directly included in the application, instead it is contained in special opcode module files. There are about twenty different opcode modules which are explicitly loaded from files and 30 more modules which are

uncompressed from a special module. This file also includes the code for the P-Code machine itself and is in fact loaded by a special trampoline. This opcode module is decompressed at runtime via the `zlib`[8] library. The 30 intrinsic opcode modules include functionality to de/encode music files, perform decryption of DRM content and carry out several cryptographic tasks, to name but a few. After the P-Code machine has been set up, these intrinsic modules are uncompressed and initialized via special handlers of the P-Code machine.

An opcode module is made of a short header with a signature indicating, that this is in fact an opcode module, and some important meta data, like an offset to the relocation table, the size of the module and its name. Figure 6 shows some important fields of the header. Just behind the header is a block of 256 random bytes. These bytes are module specific and are used to permute the assignment of opcodes and the belonging handler on a per-module basis. This basically means that opcodes have a completely different meaning across opcode modules, making it more difficult to identify opcodes when analyzing several opcode modules. Since the modules are loaded into memory by means of memory mapped files the image base address cannot be known in advance at compile time. For this reason opcode modules have a relocation table, which allows for rebasing of each module. Relocation items fall into different categories, e.g. some opcode modules have references to the C runtime which are redirected to the import address table (IAT) of the application. The remaining relocation items are simple module intrinsic calls, jumps or data offsets. Each relocation entry utilizes 8 bytes and has a 4 byte identifier which tells the rebasing algorithm the type of fix needed for the current item. The remaining 4 bytes compose the actual value to be inserted at the respective address. In addition to this, module intrinsic calls were removed beforehand, so these are fixed by this mechanism, too.



**Figure 6: Opcode Module Header**

In order to harden the protection established by the P-Code machine, opcodes taken from the opcode modules are not used directly. Instead opcodes are descrambled at runtime by means of a PRNG which is part of the P-Code machine itself. Besides that garbage data is interleaved with the actual opcodes to complicate understanding of the machine logic. To further complicate analysis all data items are stored and retrieved in an ASN.1 format, so opcode handler logic is interleaved with ASN.1 parsing code.

#### 3.3.2 Finding the Decryption Routines

The use of a P-Code machine to obfuscate program logic on the one hand and data flow on the other hand is a very

good strategy to make reverse engineering a tedious process, because existing tools at least have to be extended to be of major use. Compared to native code, analysis of a certain amount of program logic is much more tedious, because the amount of code executed to perform this very logic is much higher. In this sense the P-Code machine lowers the signal to noise ration tremendously. Especially online analysis becomes a very tedious process because one has to trace through the same (handler-)code over and over gain. So the major problem in this case was to spot the code which contributes to the decryption algorithm and the associated key setup. Possible strategies to overcome the effects of the P-Code machine could be to

1. write a custom disassembler to be able to analyze the program logic
2. use debugger scripts to trace until code writes the key to memory
3. use emulation to find the algorithm
4. use hardware breakpoints to back trace from code which accesses input data

Of course this list is not complete but rather names the most obvious ideas to overcome the protection in this case. Option 1 seems to be the most expensive strategy especially in this case because of the high number of opcodes and the complexity of the high level handlers, which would need to be fully understood in order to create a meaningful disassembly listing. Besides that the whole opcode randomization algorithm would have to be reassembled, too. The second option is extremely slow since tracing consumes a fairly amount of CPU resources, although some techniques have been researched trying to overcome this restriction[9]. The third solution in contrast provides reasonable speed and a very high level of flexibility, because obviously every single CPU feature can be controlled by using emulation[10, 11], and could be rated as the most elegant strategy. The strategy used in this case makes use of the debug registers in order to track code which accesses data read from a DRM protected music file. By using this technique it is very easy to break directly at the decryption algorithm, which is a simple DES in CBC mode[1]. It turned out that this decryption routine was in fact one of the high level handlers, i.e. it was implemented in native code, so it could be easily reverse engineered. Besides knowing the decryption algorithm itself it is of course essential to be able to reproduce the key setup. Any DRM protected file is decrypted in chunks of 0x1800 bytes. In every decryption pass the key setup and the key itself are destroyed after decryption of the respective file buffer, i.e. both data structures are overwritten with zeros. Since both data structures are dynamically allocated in each pass, the use of BMPs is not suitable for finding the key setup, because the address of the certain buffer is unknown in advance. The P-Code machine manages memory allocations similarly to heap implementations used in high level languages such as C/C++, i.e. there are multiple lists of memory chunks of different sizes. This memory management system is particularly used by the P-Code machine to allow the programs running inside the machine to dynamically allocate memory. Moreover the routines for decompressing the opcode modules also make use of this memory

management system. So whenever a new buffer for the key setup is allocated, control flow will go through the memory management function, which obviously needs to receive the desired size of the memory block as a parameter. For a single DES key setup this size is always 0x80 bytes. Finding the key setup can then be easily achieved by just setting a conditional breakpoint inside the memory management function and finally using a BPM to trace write operations to this buffer in order to break right inside the routine performing the actual key setup algorithm. The last step is now to trace all input data the key setup algorithm uses to derive the actual decryption key. Again this is no problem because hardware breakpoints can be used to spot relevant code.

## 4. DECRYPTING THE CONTENT

Due to legal issues this section has been intentionally left blank.

## 5. CONCLUSION

On the whole the DRM system offers pretty good protection mechanisms both against offline reverse engineering and against debugging. Anyhow some flaws do exist which made the process of breaking the whole system easier than it should have been. For one the usage of the debug registers to block any attempts to easily trace memory access is an effective technique, for another breaking this protection could have been much harder if the debug registers actually would have been used to set hardware breakpoints, so control flow would have depended on the BPMs firing. In this way an emulation would have been impossible and reclaiming the debug registers would have required much more intrusive measures such as patching of the protection code inside the DRM itself. Another very obvious flaw is the weak debugger detection, which only relied upon the debug flag in the PEB, which of course can be trivially patched out, and the use of fake exceptions. Many much more elaborate techniques for debugger detection exist.

Using mechanisms like Virtual Machines to carry out the core protection algorithms is a very good technique and will probably become more important in newer protection mechanisms[12, 13]. The complexity of the P-Code machine in this case could be defeated by the use of the reclaimed debug registers. In case the decryption algorithm and its associated key setup would have been emulated by the virtual CPU, this approach would have been infeasible. On the other hand this would have meant a fair increase in development time and complexity while designing the protection. It is quite evident that the number of ideas one can think of to make the process of reverse engineering more difficult is only limited by creativity and in the end every concept fundamentally based on a software protection mechanism can and probably will be broken.

## APPENDIX

### A. REFERENCES

- [1] Scott A. Vanstone Alfred J. Menezes, Paul C. van Oorschot. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [2] Hex-Rays. *IDA Pro*. <http://www.hex-rays.com/idapro/>.
- [3] Matt Pietrek. *A Crash Course on the Depths of Win32 Structured Exception Handling*.

- <http://www.microsoft.com/msj/0197/exception/exception.aspx>.
- [4] Nicolas Falliere. *Anti debugging techniques*.  
<http://www.securityfocus.com/infocus/1893>.
  - [5] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide Part 1*. <http://www.intel.com/products/processor/manuals/>.
  - [6] Jeffrey M. Richter. *Programming Applications for Microsoft Windows (Microsoft Programming Series)*. Microsoft Press Books, 1999.
  - [7] Microsoft Research. *Detours*.  
<http://research.microsoft.com/sn/detours/>.
  - [8] zlib. *zlib library*. <http://www.zlib.net/>.
  - [9] McAfee. *umss: efficient single stepping on Win32*.  
<http://www.avertlabs.com/research/blog/?p=140>.
  - [10] Cody Pierce. *PyEmu: A Multi-Purpose Scriptable x86 Emulator*.  
<http://dvlabs.tippingpoint.com/appearances/>.
  - [11] Jeremy Cooper Chris Eagle. *The x86 Emulator plugin for IDAPro*. <http://ida-x86emu.sourceforge.net/>.
  - [12] Rolf Rolles. *Defeating HyperUnpackMe2 With an IDA Processor Module*.  
[https://www.openrce.org/articles/full\\_view/28](https://www.openrce.org/articles/full_view/28).
  - [13] Benjamin Jun Carter Laren Nate Lawson Paul Kocher, Joshua Jaffe. *Self-protecting digital content*. <http://www.cryptography.com/resources/whitepapers/SelfProtectingContent.pdf>.