CodeBreakers Magazine
Security & Anti-Security - Attack & Defense
Volume 1, Issue 1, 2006

Coding Smart and Dynamic Code

For better protections and for the art of it.

+Q

March 2006

Abstract

Running code is beautiful, isn't it? Small islands of code floating in a vast sea of data. Those code islands must run in a strict, pre-defined way in order to work. Think what happens when you break into Softice. It's like freezing rain drops in mid air, while its raining on a vast, green, rain forest. Now imagine you could not only stop those rain drops, but also make them go upwards!

Introduction

Running code is beautiful, isn't it? Small islands of code floating in a vast sea of data. Those code islands must run in a strict, pre-defined way in order to work. Normally you would have dozens of files in memory, and even more threads, running on a single CPU. The outcome can be a cool 3D FPS, a MP3 player or even a word processor. The sheer complexity is mind blowing. Think what happens when you break into Softice. It's like freezing rain drops in mid air, while it's raining on a vast, green, rain forest. Now imagine you could not only stop those rain drops, but also make them go upwards! Imagine you could create thunder and lightning at your will... Its possible! In this article we are going to explore both old and new techniques of code manipulation. Dynamic code has a great value to both low level programming and protections alike. Here I wish to show you how it can be done.

Pre Build Power – macros

Where would we be without them? Hash functions would take years to write, code would take an un-usual amount of space, and the term being lazy would not reach the vast corners it reaches now. There's something very elegant in macros. Take a look at this example:

```
#define EXP8(X) EXP7(X), EXP7(128+X)
#define EXP7(X) EXP6(X), EXP6( 64+X)
#define EXP6(X) EXP5(X), EXP5( 32+X)
#define EXP5(X) EXP4(X), EXP4( 16+X)
#define EXP4(X) EXP3(X), EXP3(  8+X)
#define EXP3(X) EXP2(X), EXP2(  4+X)
#define EXP2(X) EXP1(X), EXP1(  2+X)
#define EXP1(X) EXP0(X), EXP0(  1+X)
#define EXP0(X) ((4*(X)+1)*((X)+1) & 0xff)

static unsigned char hash_table[256]={EXP8(0)};
```

hash_table is automatically filled by the pre-compiler with all byte values 0-255 in a permuted order.

Macros are well implemented in the C++ environment, but they are much more powerful in MASM environment. Just peek at MASM's reference at the keywords as MACRO, MACRO LOCAL, FOR (IRP) and FORC (IRPC). Amazingly, MASM's pre-compiler lets you define local variables inside the macro, just like in a normal function. Logical and mathematical operations on the variables are supported, and not only that, but you can also work on each byte from the argument separately! Macros let you define data and function names dynamically, and you can even define macros inside macros. Macros open the door to any cool thing you can think of!

With macros alone we can write dynamic data - a data array that will be automatically encrypted by the pre-compiler. Let's see what we need to do in order to achieve it:

```
// Define a macro that takes an
// entire string as argument
 EncryptText MACRO text:req
       LOCAL cipherByte
       cipherByte = 0AAh

// Go through each byte of the string
       FORC plainByte, <&text>

// Encrypt it
         cipherByte = cipherByte xor
'&plainByte'

// And dump the result as data byte
         db cipherByte

       ENDM
       db 0
ENDM
```

Now in our source code the data is defined with the macro:

```
EncryptText < Sin without deceivers
              /A God with no believers >
```

And on the compiled version this data is already encrypted. Of course in run time we'll

have to decrypt it back to plaintext, but half the job is done by the pre-compiler. Macros are very efficient data manipulators.

Post Build Power - .map file and the external patcher

Dynamic code is a bit more problematic. The problem is that it was never meant to be dynamic! The fact that the code is marked as a read only section is a proof of that. The more serious problem is that we don't know the final code bytes before we compile the code. Even if we write it in assembly. There's a lot of fine tuning done by the compiler and linker to make the code workable. Think of function's stack prologue and epilogue, or calling an API function, or even assembling a jump instruction. The bottom line is that the final shape and form of the code is known only after the build.

Fortunately, the linker provides all the information of the final code in the .map file. Let's consider the following problem: code integrity check. If we are going to implement an integrity check on the code, we'll need the start address and end address of the block to check. We'll also need the correct CRC value to check against. Now where are we going to get all these values? Lets consider start and end addresses of the code block. We can try to get them during pre-compile time. For example, if we have a project with only one source file to compile, we'll also have one .obj file, which means that the order of the functions in the source code will also appear in the final .exe file. CRCing one function is also a simple deal - just put `BlockStartPtr:` above the function, and `BlockEndPtr:` just below it, and send them to the CRC function. BUT, what are we going to do when we have more than one source file? Or if we want to check sum the entire application? And how are we going to get the correct CRC value, and store it back in the code, anyway?

The remedy is a new tool we'll build specifically designed for our work. An external patcher program that will read the application, manipulate or checksum whatever code we want and even send post-build information back to the application. Its all in our hands now!

The external patcher needs to communicate with the application, and for this we'll write a special structure:

In Assembly:

```
EP_INTERFACE_MAGIC         equ    49494949h
EP_TARGET_INTERFACE STRUCT
                     _interfaceFlag      DWORD
EP_INTERFACE_MAGIC
                     _crcBlockStart      DWORD ?
                     _crcBlockEnd DWORD ?
                     _crcCorrectValue    DWORD ?
EP_TARGET_INTERFACE ENDS
```

Or in C++:

```
#define EP_INTERFACE_MAGIC       (0x49494949)

#pragma pack(push, 1) // no data structure
                      // alignment
struct EP_TARGET_INTERFACE
       {
              DWORD        _interfaceFlag;
              DWORD        _crcBlockStart;
              DWORD        _crcBlockEnd;
              DWORD        _crcCorrectValue;
       };
#pragma pack(pop)
```

Please notice the `_interfaceFlag` dword. The external patcher needs to locate the structure in the file, so this flag, initialized with an un-common magic dword, marks the start of the interface. To accomplish code integrity check, the external patcher grabs both the application's .exe file and .map file. From the .map file our patcher calculates the application's image start and end addresses - it can be the entire image, (the entire .text section) or it can be a specific function. Next the external patcher opens the .exe file, and locates the interface with the aid of the flag dword. The external patcher performs the check sum of the specified code block, and stores both addresses and the correct result back to the exe, in the interface.

At run time, the application grabs these values, which are now valid, and performs the integrity check! Please note that the addresses should be converted from RVA, and proper error handling should be implemented, but the basic idea is that simple to implement with the aid of the external patcher.

Combining powers - Dynamic Code

Why stop here? Let's work with both pre-build and post-build powers together! Not always when you add two great things, you get an even greater combination. I mean, if you play Beethoven and Mozart together you usually get a classical noise. But not so in our case. Combining macros and post build patcher is a beautiful and amazingly powerful technique. As an example we will write a function that automatically locks and unlocks itself. Let's see what we need:

- The code block should be unlocked at runtime. So we'll have a loop that encrypts/decrypts the code. The decryptor will execute at the function's prologue, while the encryptor will execute at the function's epilogue.
- The code block should be locked in the final .exe file. This means the external patcher will have to lock the block after compilation. Which also means we'll have a flag at the block's first opcode, so the external patcher will be able to locate the block.

Let's start with the macro:

```
AUTO_LOCK_BEGIN MACRO
LOCAL blockStart, blockEnd, lockLoop

;; init
pusha
mov esi, offset blockStart
mov edi, esi
mov ecx, (offset blockEnd - offset blockStart)

;; decrypt
lockLoop:
lodsb
xor al, 55h
stosb
loop lockLoop

;; goto the beginning of the block
popa
jmp blockStart
      ;; information for the external patcher:
dd EP_AUTO_LOCK_MAGIC
                              ; flag
dd (offset blockEnd - offset blockStart)
                              ; block size

blockStart LABEL BYTE

AUTO_LOCK_END MACRO
blockEnd LABEL BYTE

;; init
pusha
mov esi, offset blockStart
mov edi, esi
mov ecx, (offset blockEnd - offset blockStart)

;; encrypt
LockLoop:
lodsb
xor al, 55h
stosb
loop lockLoop

;; done
popa
ENDM
ENDM
```

Please notice how AUTO_LOCK_END macro is defined inside AUTO_LOCK_BEGIN macro. This means that AUTO_LOCK_END can use all the variables declared in the parent macro. This is how these macros are used:

```
ReallyCoolFunctionThatDoesNothingButBragItsProte
ction
    PROC
       AUTO_LOCK_BEGIN
       ;; write what ever we want here...
       ;; guess what this does ;)
       mov     esi,edx
       xor     cl,cl
       shld    edx,eax,1
       shld    esi,eax,2
       adc     cl,0
       xor     edx,esi
       xchg    eax,edx
       xor     dl,cl
       AUTO_LOCK_END
       ret
ReallyCoolFunctionThatDoesNothingButBragItsProte
ction
     ENDP
```

The external patcher's job is to locate this code block, use the given block size information, and encrypt the code. If there are more than one block in the file, there will simply be more flags. The external patcher will go through the entire file, and locate all these flags.

Here's a cool optimization to the macro above, that will use the same decryption loop for both encryption and decryption:

```
AUTO_LOCK_BEGIN MACRO
LOCAL blockStart, blockEnd, lockLoop, lockMain,
                    retJunction

;; init
lockMain:
       pusha
       mov esi, offset blockStart
       mov edi, esi
       mov ecx, (offset blockEnd - offset
blockStart)

;; decrypt
lockLoop:
       lodsb
       xor al, 55h
       stosb
       loop lockLoop

       ;; goto the beginning of the block
       popa

       ;; use SMC to toggle RET / JMP
       ;;instructions
       xor byte ptr [retJunction], 028h

       ;; sometimes its jump, sometimes ret...
       retJunction:
       db 0C3h, 8

       ;; information for the external patcher:
       dd EP_AUTO_LOCK_MAGIC            ; flag
       dd (offset blockEnd - offset blockStart)
       ; block size

       blockStart LABEL BYTE

       AUTO_LOCK_END MACRO
              blockEnd LABEL BYTE

              ;; encrypt
              call lockMain
       ENDM
ENDM
```

Run Time Power - "The Running Line"

In an article titled "Anti Debugging Tricks" posted back in 1994 a technique called "The Running Line" was described (apparently, it was first published by Serge Pachkovksy, but i couldn't find it). This technique reveals a self tracing, self modifying code. The idea is really beautiful. Basically, there's a tracer function that's called after every "normal" instruction. On this tracer function, you could change the code dynamically at run-time and return to it. In practice, exception handling and the trap flag are used. The CPU raises the "single step" exception whenever the trap flag is set. Under DOS this means that int1 is raised. A self-tracing program would take advantage of this feature, and simply by changing the int1 handler and setting the trap flag, you would have a self-tracing application.

Fortunately, this technique is also applicable under windows. Only this time there's no int1, but a normal exception, and a normal exception handler. This technique is extremely efficient against debuggers and disassemblers. SoftIce messes the trap flag, so you can't single step into a self-tracing function, and under a disassembler a naive code block might change in run time, thanks to the running line handler. Here's a simple tracer that changes one of the code instructions:

```
;; SEH macros
SEH_NODE STRUCT
       _prevHandler  DWORD ?
       _exceptionHandler   DWORD ?
SEH_NODE ENDS

PUSH_SEH MACRO sehHandler:req
       ASSUME FS:NOTHING
       mov eax, fs:[0]
       ASSUME eax:ptr SEH_NODE
       push sehHandler
       push [eax]._exceptionHandler
       mov fs:[0], esp
ENDM

POP_SEH MACRO
       pop fs:[0]
       add esp, 4
ENDM
```

```
;; exception handler
ExpHandler PROC c expRecord:DWORD,
expFrame:DWORD,
contextPtr:DWORD, dispContext:DWORD
       pusha
       mov ebx, contextPtr
       ASSUME ebx:ptr CONTEXT

       ;; clear trap flag
       and [ebx].regFlag, 0FFFFFEFFh

       ;; change the opcode to NOP
       mov ebx, [ebx].regEip
       mov byte ptr [ebx], 90h

       popa
       mov eax, ExceptionContinueExecution
       ret
ExpHandler ENDP

;; Self tracing function
SelfTracingCode PROC

       ;; set up the handler
       PUSH_SEH

       ;; set the trap flag
       pushf
       or byte ptr [esp+1], 1
       popf

       ;; this will not be traced
       xor eax, eax
       ;; endless loop - this code
       ;;will change at run-time
       JMP $

       ;; remove the handler
       POP_SEH

       ret
SelfTracingCode ENDP
```

The possibilities of the running line technique are endless. The tracer is like an invisible storm that can change the code at run time. Self modifying, self decompressing and even self compiling code is no fiction, it can be done!

This is only the beginning...

You can probably guess what I am going to say... "Why stop here?" Good question! :) Let's combine all the above techniques! The final example we are going to explore is one funky chicken. We are going to write code that runs backwards.

In theory it is very simple, we'll use the running line handler to re-set the EIP register one instruction upwards, and then restore execution for one more step. When this is done on a block of opcodes, we'll have a whole block that actually runs backwards. In practice, if we want to re-set the EIP register one instruction upwards we'll have to know the size of the opcode, and decrease this value from EIP register. Getting the opcode size is simple:

```
       opcodeStart:
       xor eax, eax
       db ($ - offset opcodeStart)
```

And there, we have the opcode, followed by its size. Doing this for all the opcodes in the block can be a real pain, so we'll do it in a macro:

```
OPREVBEGIN MACRO
       LOCAL opcodeStart, opcodeEnd

       ;; dump the size
       db (offset opcodeEnd - offset
opcodeStart)

       opcodeStart LABEL BYTE

       OPREVEND MACRO
              opcodeEnd LABEL BYTE
       ENDM
ENDM

;; wrap the opcode with the macro above
R MACRO opcode:req
       OPREVBEGIN
       opcode
       OPREVEND
ENDM

;; a block of opcodes, each one has its size
;; attached
R< rol eax, 10 >
R< mov ecx, "hi, " >
R< xor edx, ecx >
R< add eax, "man!" >
R< stosd >
```

Next we'll have to make things simpler to the external patcher. We should supply the block's beginning address, and its total size. While we are at it, we'll make things simpler to the running line handler:

```
REV_BLOCK_BEGIN MACRO
LOCAL blockStart, blockEnd

;; set up exception handler to RevHandler
function
PUSH_SEH

;; make sure ebp points to opcodes sizes array
mov ebp, (offset blockStart)

;; set trap flag
pushf
or byte ptr [esp+1], 1
popf

;; this runs backwards, so jump to the end =)
jmp blockEnd

;; information for the external patcher:
dd EP_REVERSECODE_MAGIC                  ; flag
dd (offset blockEnd - offset blockStart)
                                  ; block size

blockStart LABEL BYTE
REV_BLOCK_END MACRO
blockEnd LABEL BYTE

;; remove exception handler
POP_SEH
ENDM
ENDM
```

In the source code, this is how a block that runs backwards is implemented:

```
CoolFunctionThatRunsBackwards PROC
       ;; mark the beginning
       REV_BLOCK_BEGIN

       ;; every opcode here should
       ;;have its size attached
       R< ror eax, 10 >
       R< mov ecx, "what" >
       R< xor edx, ecx >
       R< add eax, "s up" >
       R< nop >

       ;; mark the end
       REV_BLOCK_END
       ret
CoolFunctionThatRunsBackwards ENDP
```

Let's order things a bit. This is the above code after compilation:

```
{ prologue code ;
 < OP_1 size, OP_1 >,
< OP_2 size, OP_2 >,
 .
 .
 < OP_N size, OP_N >,
 ; epilogue code }
```

The external patcher's job is post build code manipulation. First it finds the block with the aid of the flag. Next it moves all the sizes to the start of the block, and finally it sorts all the opcodes one after the other in a reversed order. This is how it will look after the external patcher's job:

```
{ prologue code ;
 < OP_1 size >,
 < OP_2 size >,
 .
 .
 < OP_N size >,
 < OP_N >,
 < OP_N-1 >,
 .
 .
 < OP_2 >,
 < OP_1 >,
 ; epilogue code }
```

The tracer's job is to grab the opcode's size from the array, and decrease it from EIP register. The output of this application is a full blown function that runs backwards! Sure, it has its disadvantages, like not supporting jump / loop instructions, but it beats writing a full disassembler to accomplish this task.

Final Words

What a rush, ahh? Just imagine what more we can do with these techniques... For example, we can even combine both AUTO_LOCK and REV_BLOCK on the same function, one on top the other. If we make sure the external patcher first processes the REV_BLOCK code and then processes AUTO_LOCK code, then at run time this function will unlock itself, then execute backwards and finally lock it self back!

Personally I would like to thank 29A guys for their magazine and wonderful low level work. I would also like to thank whoever wrote "assemblur" crackme (search at crackmes.de under DOS crackmes). This crackme is a beautiful piece of work, and it gave me many ideas about how far dynamic code can go.