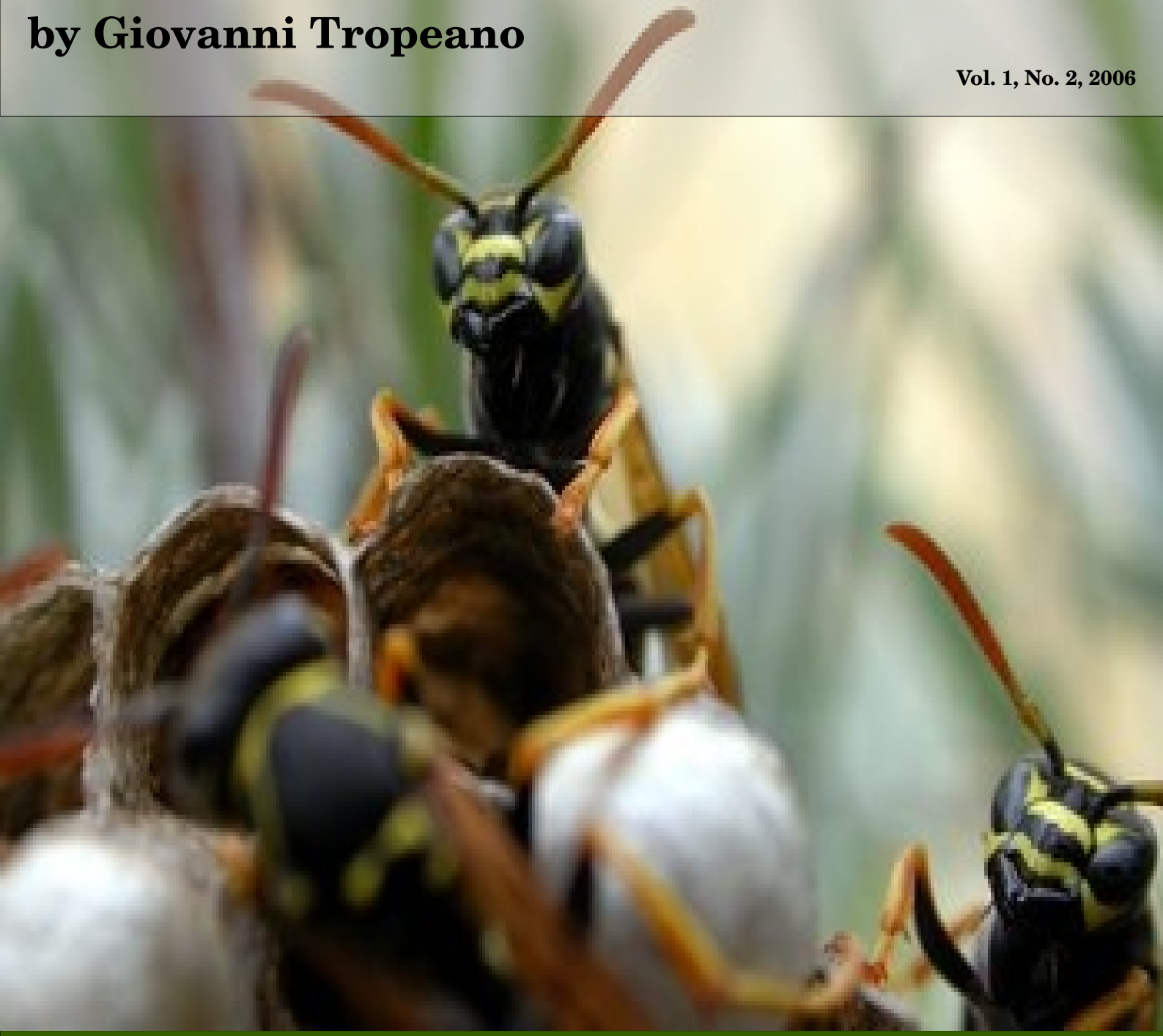
CodeBreakers Magazine
Security & Anti-Security - Attack & Defense

Self Modifying Code

by Giovanni Tropeano

Vol. 1, No. 2, 2006



Abstract:

This article takes an in depth look at self modifying code (SMC) and how you can use it in your own applications. There are examples in C++ using inline assembly, as well as pure assembler. I also talk about executing code on the stack, which is essential to successfully write and execute SMC.

Self Modifying Code

By Giovanni Tropeano

This article takes an in depth look at self modifying code (SMC) and how you can use it in your own applications. There are examples in C++ using inline assembly, as well as pure assembler. I also talk about executing code on the stack, which is essential to successfully write and execute SMC.

1 Preface

Ok, let's get started. This document is probably going to be long, as I want to make sure I explain everything as clearly as I possibly can. It's my interpretation of a million articles and hundreds of hours of writing SMC. . . I'll try to cram it all in here. So grab yourself a beer (or whatever your beverage of choice may be), turn up the tunes, and get ready to learn how to avoid the casual crackers from wreaking havoc on your apps! Along the way I'll teach you about Windows memory, and a few other things you may not know.

2 Brief History of Self Modifying Code

Back in the day, programmers had the luxury of using self modifying code at will. MS-DOS programmers used it at will - any serious attempt at protecting applications a 10 or 20 years ago involved SMC (self modifying code). Even compilers used it that compiled code into memory.

Then in the mid 90's something happened. It was called Windows 95/NT. All of a sudden us programmers had to think about all that shit we learned in protecting our apps and somehow move it over to this new platform. All the tricks we learned had to be forgotten, as we no longer had uncontrolled access to memory, hardware, and the general operating system itself. People started to think that writing SMC was impossible unless you used VxD, which Windows is notorious for not documenting well. Then it was dicovered that we COULD still use SMC in our apps. One of two ways would work just fine for us - either use the WriteMemoryProcess export of Kernell32 or put the code on the stack to be modified.

The rest of this document will mostly deal with Microsoft Visual C++ and 32bit executable code.

3 Windows Memory - How it's put together

Creating SMC in Windows isn't as straight forward as I would of hoped. You're going to have to deal with a few quirks and kinks that Windows will throw your way. Why? 'Cause it's Microsoft.

As you probably already know, Windows allocated 4 Gigs of virtual memory for a process. To address this memory, Windows uses two selectors. One is loaded in the CS Segment register, and the other one is thrown into the DS, SS, and ES registers. They all use the same base address of memory (equal to zero) and have the same limits of 4 Gigs.

There's only ONE segment that contains both code and data, as well as the stack of a process. You can use a NEAR call or a jump to control code located in the stack. You don't have to use the SS to access the stack, either. Although the value of the CS register is not equal to the value of the DS, ES, and SS registers, the MOV dest, CS:[src], MOV dest, DS:[src], and MOV dest, SS: [src] instructions address the same memory location.

Memory areas (pages) containing data, code, and the stack have different attributes. For instance, code pages will allow reading and executing, data pages - reading and writing, and the stack - reading, writing, and executing simultaneously.

There's also some security attributes attached to each of those, but I'll explain a bit more on that later as we need it.

4 Using WriteProcessMemory - New Best Friend

The easiest way (in my opinion) to change some bytes of a process is to use the WriteMemoryProcess (as long as some security flags are not set).

The first thing we want to do on the memory process we want to modify, is open it with the OpenProcess function, with the PROCESS_VM_OPERATION and PROCESS_VM_WRITE access attributes.

Here's a simple example of some SMC, that we will talk about. We will need to use some inline assembly to accomplish this in C++. This of course can be done in all assembly, but not in this document. It's going to be long enough!

Listing 1: Using WriteProcessMemory to Create SMC

```
int WriteMe(void *addr, int wb)
{
    HANDLE
h=OpenProcess(PROCESS_VM_OPERATION|
PROCESS_VM_WRITE,
    true, GetCurrentProcessId());
    return WriteProcessMemory(h, addr, &wb, 1, NULL);

}

int main(int argc, char* argv[])
{
    _asm {
        push 0x74 ; JMP --> > JZ
        push offset Here
        call WriteMe
        add esp, 8
    Here: JMP short here
    }
    printf("Holy Sh^& OsIX, it worked! #JMP SHORT $-2
was changed to JZ $-2n");
    return 0;
}
```

As you can see, the program is replacing this infinate jump with a valid JZ instruction. This allows the program to continue, and we see the message telling us the jump was changed. Cool, huh? I bet by now you are thinking...hmmm, I wonder if I could do "this" or "that"? Odds are, probably!

There are some drawbacks of this (the WriteMemoryProcess). First of all, the experienced cracker WILL recognize this in the imports table. He will most likely set a breakpoint on this call, and then from there step through only the code he wants to. Using WriteProcessMemory is only reasonable in compilers that compile into memory, or in unpackers of executable files, but you sure CAN use it to throw off the casual cracker. I have used this in my apps a lot.

Another thing that sucks about WriteMemoryProcess is the inability to create new pages in memory. It only works on existing pages. There are other ways to accomplish this, but we'll look at executing code on the stack as our option.

5 Putting Code on the Stack, and executing it!

Executing code on the stack is not only possible, it is necessary. It makes life easy on compliler so they can generate code. But doesn't this pose a security threat? You bet your ass it does. But there's not much you can do as a programmer because if for instance, you were to install a patch that don't allow code to be executed on the stack - odds are most of your programs would not run! Linux has such a patch, and so does Solaris, although it's a good guess there's only like 2 people who installed them (the authors hee hee).

Remember those drawbacks I just talked about using WriteMemoryProcess? Well, those are overcome using the stack to execute code for a couple of reasons. One is, it's almost impossible for a cracker to trace the instructions that modify an unknown memory location. He'll have to work his butt off to analyize the protection code, and he probably won't have much success! The other reason executing code on the stack is a positive thing is, at any moment, the application may allocate as much memory for the stack as it sees fit, and then, when it becomes unnecessary, free that space. By default, the system allocates 1 MB of memory for the stack. If this memory appears to be insufficient to solve the task, the necessary quantity can be specified when the program is configured.

There's some specific stuff you need to know about executing stuff on the stack... so let's get to it in the next section.

6 Why relocatable code can be bad for your health

You have to be aware that the location of the stack is different on Windows 9X, Windows NT, and Windows 2000. To make sure your program works when it moved from one system to another requires it to be relocatable. Achieving this is not hard, you just have to follow a few simple rules - yes damn rules!

Fortunately for us, in the 80x86 world all short jumps and near calls are relative. That means it don't use linear addresses, but the difference between the target address and the next instruction. This sure does simplify our life when making relocatable code, but it also has some restrictions.

For example, what happens if the void OSIXDemo() {printf("Hi from OSIXn");} function is copied to the stack, and control is passed to it? Since the address of printf has changed, this will most likely result an error!

In assembler, we can easily fix that by using register addressing. A relocatable call of the printf function may look simplistic, for example LEA EAX, printfNCALL EAX. Now the ABSOLUTE linear address, not a relative one, will be placed in the EAX register. Now it don't matter where it's called from, control will still be passed to printf.

Doing such things requires that your compiler support inline assembly. I know, this sucks if you're not

interested in lower level instruction code, and it CAN be achieved using only high level languages. Here's a quick example:

Listing 2: How a Function Is Copied to and Executed in the Stack

```
void Demo(int (*_printf) (const char *,...))
{
    _printf("Hello, OSIX!n");
    return;
}

int main(int argc, char* argv[])
{
    char buff[1000];
    int (*_printf) (const char *,...);
    int (*_main) (int, char **);
    void (*_Demo) (int (*) (const char *,...));
    _printf=printf;

    int func_len = (unsigned int) _main - (unsigned int)
_Demo;
    for (int a=0; a<func_len; a++)
    buff[a] = ((char *) _Demo)[a];
    _Demo = (void (*) (int (*) (const char *,...))) &buff[0];

    _Demo(_printf);
    return 0;
}
```

So don't let anyone tell you executing code on the stack is not possible using a high level language.

7 I got your optimization right here!

You will need to really think about what compiler you are going to use, and study some details regarding that compiler if you plan on using SMC or executing code on the stack. In most cases, the code of a function WILL FAIL on the first attempt when executing on the stack, especially if your compiler is set to "optimize".

Why does this happen? Because in pure high level languages such as C or Pascal, it is damn near impossible to copy the code of a function to the stack or elsewhere. The programmer may obtain the pointer to a function, but there's no standard on how to interpret it. Us programmers call this, the "Magic Number" and it is known only to the compiler.

Fortunately, almost all compilers use the same logic to genratate code. This allows the program to make certain assumptions about the compiled code. The programmer also is able to make certain assumptions.

Let's take a look back at Listing 2. We assume that the pointer to this function coincides with the beginning of the function, and that the body is located behind the beginning. Most compilers will use this sort of "common sense compiling" but don't count on all of them doing that. The big guys follow this rule though (VC++, Borland, etc). So unless you're using some unknown or new compiler, don't worry about it. One note about VC++: if you are in debug mode, the compiler will insert an "adapter" and allocate the functions somewhere else. Damn Microsoft. But no worries, just make sure the "Link Incrementally" box is checked in the options, and you can force good code to be generated. If your compiler don't have this option or something similar, you can either NOT use SMC, or use another compiler!

Another problem is to determine the length of a function. Doing this reliably is a bit of a trick, but can be done. In C(++) the sizeof instruction doesn't return the length of the function itself, but the size of the pointer to the function. But as a rule, compilers allocate memory according to the order they appear in the source code. So... the length of the body of a function is equal to the difference between the pointer to the function and the pointer to the function following it. Easy! Remember though, optimizing compilers DO NOT follow these procedures and the method I just described will not work. See why optimizing compilers are bad for your health if you are writing SMC?!?!?

And yet another thing that optimizing compilers will do is delete variable that they THINK are not being used. Going back to our example in Listing 2, something is written to the buff buffer, but nothing is READ from that place. Most compilers are unable to recognize that control was passed to the buffer, so they delete the copying code. The bastards! That's why control is passed to the unitialized buffer, and then...boom. Crash. If this problem arises, clear the "Global optimization" checkbox and you'll be okay.

If your program STILL does not work, don't give up. The reason is probable the compiler inserting the call of a routine that monitors the stack into the end of each function. Microsoft's VC++ does this. It places the call of the function __chkesp into debugged projects. Don't bother looking it up in the documentation - there is none - imagine that! This call is relative, and there is no way of disabling it. However, in your final project, VC++ doesn't inspect the state of the stack when exiting a function, and stuff will work smoothly.

8 Using SMC in your own apps

Ok finally here - the section you have all been wanting to get to. If you have made it this far, I applaud you. (clap clap)

Ok, by now you may be asking yourself (or asking me) "What's the benefit of executing code (a function) on the stack?" And the answer is (drumroll please...) The code of a function on the stack can be changed on the fly, at will - such as decrypting! And the crowd says Ahhhhhhhhhhhhhh.

The encrypted code makes it a pain in the ass for the cracker to disassemble. Of course, with a debugger this gets a little easier, but still makes his/her life difficult.

The simplest encrypting algorithm will sequentiall process each line of code using the exclusive OR operation (XOR). And running this again will produce our original code!

Here's an example that reads the contents of our DEMO function, encrpypts it, and writes the results into a file.

Listing 3: How to Encrypt the Demo Function

```
void _bild()
{
    FILE *f;
    char buff[1000];
    void (*_Demo) (int (*) (const char *,...));
    void (*_Bild) ();
    _Demo=Demo;
    _Bild=_bild;

    int func_len = (unsigned int) _Bild - (unsigned int)
_Demo;
    f=fopen("Demo32.bin", "wb");
    for (int a=0; a<func_len; a++)
    fputc(((int) buff[a]) ^ 0x77, f);
    fclose(f);
}
```

After it has been encrypted, the contents are then placed into a string variable. Now the Demo function can be removed from the initial code. Then when we need it, it may be decrypted, copied into the local buffer, and called for execution! Kick ass huh?

Here's an example of how we would impliment that:

Listing 4: The Encrypted Program

```
int main(int argc, char* argv[])
{
    char buff[1000];
    int (*_printf) (const char *,...);
    void (*_Demo) (int (*) (const char *,...));
    char code[]="x22xFCx9BxF4x9Bx67xB1x32x87
x3FxB1x32x86x12xB1x32x85x1BxB1
x32x84x1BxB1x32x83x18xB1x32x82
x5BxB1x32x81x57xB1x32x80x20xB1
x32x8Fx18xB1x32x8Ex05xB1x32x8D
x1BxB1x32x8Cx13xB1x32x8Bx56xB1
x32x8Ax7DxB1x32x89x77xFAx32x87
x27x88x22x7FxF4xB3x73xFCx92x2A
xB4";

    _printf=printf;
    int code_size=strlen(&code[0]);
    strcpy(&buff[0], &code[0]);

    for (int a=0; a<code_size; a++)
    buff[a] = buff[a] ^ 0x77;
    _Demo = (void (*) (int (*) (const char *,...))) &buff[0];
    _Demo(_printf);
    return 0;
}
```

Note that the printf function displays a greeting. At first glance you may not notice anything unusual, but look at where the string "Hello, OSIX!" is located. It should not be in the code segment (Borland puts it there for some reason) - therefore check the data segment and you will see it - right where it ought to be!

Now, even if the cracker is looking at the source code, it's still going to be one hell of a puzzle! I use this method to conceal "secret" information all the time (serial number, ken generators for my programs, etc).

If you are going to use this method to verify a serial number, the verification method should be organized so that even when decrpyted, it will still puzzle the cracker. I'll show you such an example in the next listing.

Remember, when implimenting SMC you need to know the EXACT location of the bytes you are trying to change. Therefore, an assembler should be used instead of a high level language. C'mon, stay with me, we are almost done!

There is one problem connected with using assembler to do this - do change a certain byte the MOV instruction needs to be passed the ABSOLUTE linear address (which you have probably already figured out is UNKNOWN before compiling). BUT.. we can find this info out in the course of running the program. The CALL $+5POP REGMOV [reg+relative_address], xx statement has gained the greatest popularity with me. It works.Inserted as the following statement, it executes the CALL instruction, and pops the return address from the stack (or the absolute address of this instruction). This is used as a base for addressing the code of the stack function.

And now that example on the serial numbers I promised you...

Listing 5: Generates a Serial Number and Runs in the Stack

```
MyFunc:
push esi ; Saving the esi register on the stack
mov esi, [esp+8] ; ESI = &username[0]
push ebx ; Saving other registers on the stack
push ecx
push edx
xor eax, eax ; Zeroing working registers
xor edx, edx
RepeatString: ; Byte-by-byte string-processing loop

lodsb ; Reading the next byte into AL
test al, al ; Has the end of the string been reached?
jz short Exit

; The value of the counter that processes 1 byte of the
string
; must be choosen so that all bits are intermixed, but
parity
; (oddness) is provided for the result of transformations
; performed by the XOR operation.

mov ecx, 21h
RepeatChar:
xor edx, eax ; Repeatedly replacing XOR with ADC
ror eax, 3
rol edx, 5
call $+5 ; EBX = EIP
pop ebx ; /
xor byte ptr [ebx-0Dh], 26h;
; This instruction provides for the loop.
; The XOR instruction is replaced with ADC.
loop RepeatChar
jmp short RepeatString

Exit:

xchg eax, edx ; The result of work (ser.num) in EAX
pop edx ; Restoring the registers
pop ecx
pop ebx
pop esi
retn ; Returning from the function
```

This algorithm is kinda weird - because repeatedly calling a function and passing it the same arguments may return either the same or a completely different result! It depends upon the length of the username. If it is odd, XOR is replaced with ADC when the function is exited. If it's even,nothing similar happens!

Well...that's it for now. I hope you at least learned a little something from this document. Took me over 2 hours to type! Feedback is always welcome.

Take care, and I'd be glad to have a look at any SMC code you may send my way!