Understanding conditionals in shellcode

Author: Dr Craig S Wright GSE GSM LLM MStat

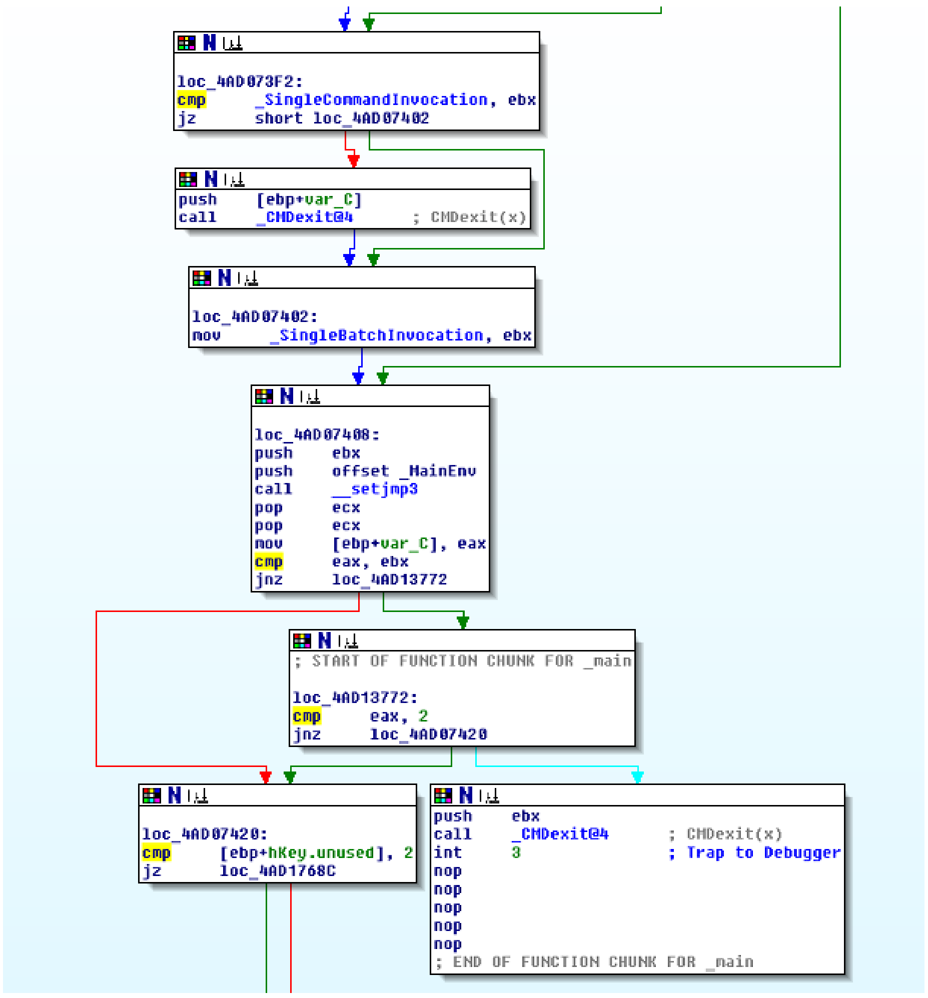
Abstract / Lead

This article is going to follow from previous articles as well as going into some of the fundamentals that you will need in order to understand the shellcode creation process. In this article, we are looking at extending our knowledge of assembly and shellcoding. This is a precursor to the actual injection and hooking process to follow. In this piece, we will investigate how you can determine code loops, the uses of loops as well as acting as an introduction into how you can reverse engineer assembly or shellcode into a higher level language and even pseudo-code, all of which forms an essential component of creating and executing one's own exploit successfully. By gaining a deep understanding just how code works and to know where to find the fundamentals shellcode programming language we hope to take the reader from a novice to being able to create and deploy their own shellcode and exploits.

Electronic copy available at: https://ssrn.com/abstract=3153489

Introduction

In the previous article, "*Beyond automated tools and Frameworks: the shellcode injection process*" we started to introduce the basic assembly functions and instructions. We will follow from previous articles and expand on the use of the fundamentals such that you can start to develop a deep understanding of the shellcode creation process. We will do this by extending our knowledge of assembly and shellcoding as a precursor to the actual injection and hooking process to follow. In this article we will investigate how you can determine code loops, the uses of loops as well as acting as an introduction into how you can reverse engineer assembly or shellcode into a higher level language and even pseudo-code, all of which forms an essential component of creating and executing one's own exploit successfully. By gaining a deep understanding just how code works and to know where to find the fundamental shellcode programming language we hope to take the reader from a novice to being able to create and deploy their own shellcode and exploits.



```
loc_4AD073F2:
cmp     _SingleCommandInvocation, ebx
jz      short loc_4AD07402
```

```
push    [ebp+var_C]
call    _CMDexit@4          ; CMDexit(x)
```

```
loc_4AD07402:
mov     _SingleBatchInvocation, ebx
```

```
loc_4AD07408:
push    ebx
push    offset _MainEnv
call    __setjmp3
pop     ecx
pop     ecx
mov     [ebp+var_C], eax
cmp     eax, ebx
jnz     loc_4AD13772
```

```
; START OF FUNCTION CHUNK FOR _main

loc_4AD13772:
cmp     eax, 2
jnz     loc_4AD07420
```

```
loc_4AD07420:
cmp     [ebp+hKey.unused], 2
jz      loc_4AD1768C
```

```
push    ebx
call    _CMDexit@4          ; CMDexit(x)
int     3                   ; Trap to Debugger
nop
nop
nop
nop
nop
; END OF FUNCTION CHUNK FOR _main
```

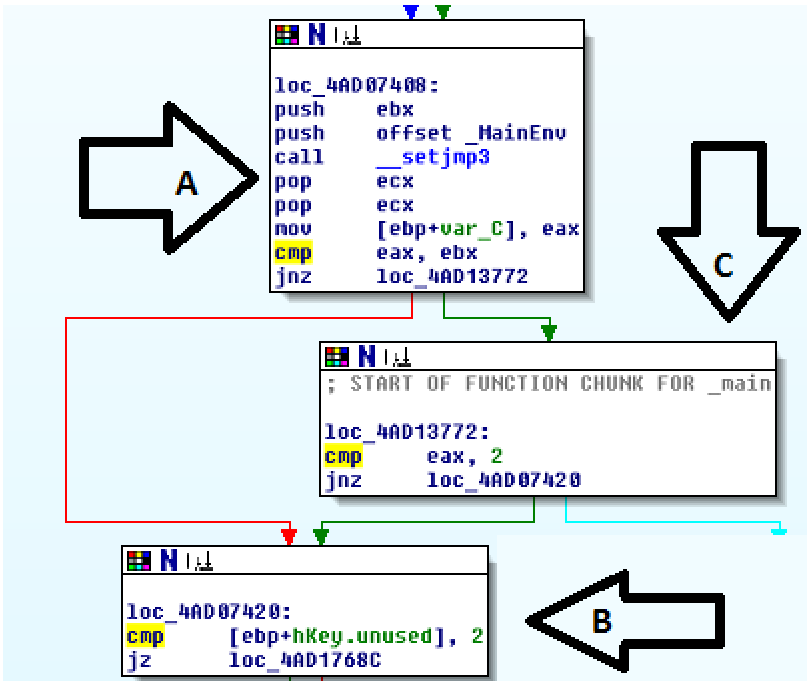
Figure 1: IDA disassembled "cmd.exe".

Using an interactive disassembler (such as IDA Pro) simplifies the process and in many cases creates a complete flow diagram and graph of the program we wish to analyse or create (Figure 1). This helps in the actual process of analysis, but does little in the manner of explanation. Unless we already possess the knowledge of why IDA set the resulting structure as it has done, we are little closer in actually being able to understand the code segment.

Electronic copy available at: https://ssrn.com/abstract=3153489

Rehashing conditionals and branching

If we take an individual jump statement in Figure 2, we see that block “A” ends in a “compare” (CMP) and “Jump if not zero” (JNZ). This is listed in “Code 1”. Here, the test “Cmp EAX, EBX” is an “implied sub” or an instruction that will evaluate the statement “EBX – EAX” without modifying the values stored in the operand, in this case the register EAX. In the event that EAX and EBX are the same, the flag “ZF” or the zero flag will be set to zero (ZF=0). If the ZF is set, the code jumps to location “B” (the green IDA arrow) and if the ZF is not set he code jumps to location “C” (the red IDA arrow).



```
loc_4AD07408:
push    ebx
push    offset _MainEnv
call    __setjmp3
pop     ecx
pop     ecx
mov     [ebp+var_C], eax
cmp     eax, ebx
jnz     loc_4AD13772
```

```
; START OF FUNCTION CHUNK FOR _main

loc_4AD13772:
cmp     eax, 2
jnz     loc_4AD07420
```

```
loc_4AD07420:
cmp     [ebp+hKey.unused], 2
jz      loc_4AD1768C
```

Figure 2:   A Jump in “cmd.exe”.

The next command in the block, “Jnz   loc_4AD13772” will evaluate the zero flag and hence we have an IF statement (Code 2). For the present, we have not looked at setting variables and constants, but in time we can do this as well through analysing the values that are being pushed and popped onto and off of the stack in segment A (Figure 2).

```
…
1      Cmp    EAX,   EBX
2      Jnz    loc_4AD13772
…
```

Code 1: Testing conditionals

We can represent this statement in pseudo-code (Code 2). From this we can see that “simplifying the shellcode into pseudo code that a human can understand easier increases the amount of code significantly. This increase comes about anytime we take low level code (such as shellcode) and convert it into a higher level language (including pseudo code).

In order to learn how to understand shellcode fully, you will need to comprehend both the creation of assembly directly as well as the reversing process (Foster, Osipov, Bhalla, & Heinen, 2005). That is, you will need to be able to take pseudo code and write shell code as well as the reverse of the process, taking shellcode and converting this into an understandable pseudo-code. The key element here of course is practice. Writing and reversing code is not as difficult as it seems at first glance; it is a matter of practice. The more time you spend writing and reversing simple statements, the better you will become.

```
If (EAX <> EBX)                  // If EAX does not equal EBX (A)
{
       Goto  loc_4AD13772        // Jump to location 4AD13772 (B)
}
Else                             // Otherwise
{
       Goto  loc_4AD07420        // Jump to location 4AD07420 (C)
}
```

Code 2: The Pseudo Code

Here we see the use of conditional statements at their most basic. These form the basis of all branching and even many loops within code.

Back to Shellcode

So, if we look again at the shellcode example we introduced in the prior article (Code 3) we see that we can create a simple graph of the code (Figure 3).

```
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x23\x41\x41\x41\x41"
"\x42\x42\x42\x42";
```

Code 3: Shellcode sample[1]

In this example we have a simple conditional statement as well as a simple function call. The conditional statement is based on the "bound" instruction that is commonly used to ensure a signed array index (16- or 32-bit register) value falls within the upper and lower bounds of a block of memory. Where the result is "true" the code branches as is displayed on the left of Figure 3 and when this fails (the result is "false"), additional operations are conducted

The jump, our conditional statement, is conducted using a *JNB* instruction. Here, the code will "*Jump if not (unsigned) below*" or the Carry Flag (CF) is set to zero (0).

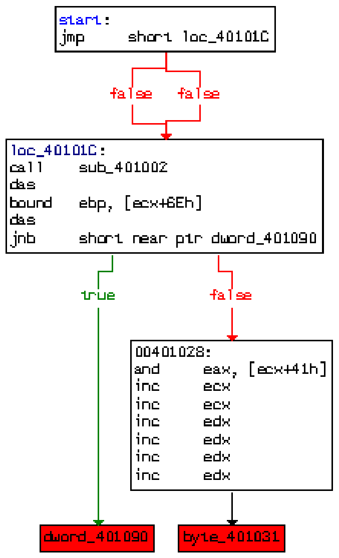
All this to execute a shell as a system instruction.

This shellcode example was extremely simple. In creating shellcode, we are generally attempting to create as small a sample as possible. There are reasons for this, as we increase the size of the code, we increase the probability that we will be detected. More, many heap spray and buffer overflow attacks are limited in the amount of information that we can send to them. This point is important. If we need 250 bytes to be able to do what we want to achieve in our shellcode and the exploit will work with a maximum size of 200 bytes, our exploit will fail.

[1] This sample of shellcode has been taken from (Zillion, 2002). This page goes into detail as to the operation of the shellcode and the reader is encouraged to step through this. The reader will find countless many examples online with a simple Google search and many good examples are also included within the Metasploit framework.

So in this case, size matters. It is just the reverse of that we commonly think of and here the smaller the code, the better it is.

We also see this when we are looking at malware. Here, the size of the code can be larger than it is in shellcode, but it is always unlikely that a 100Mb code sample would ever be installed and run as a malware exploit. That stated, stranger things do occur! In regards to malware, the use of loops, functions and conditionals are frequently used in order to obscure code and make it more difficult for an analyst to decode them. This can include simple encryption and compression routines as well as dead end code branches that are designed to simply waste the analyst's time and effort.

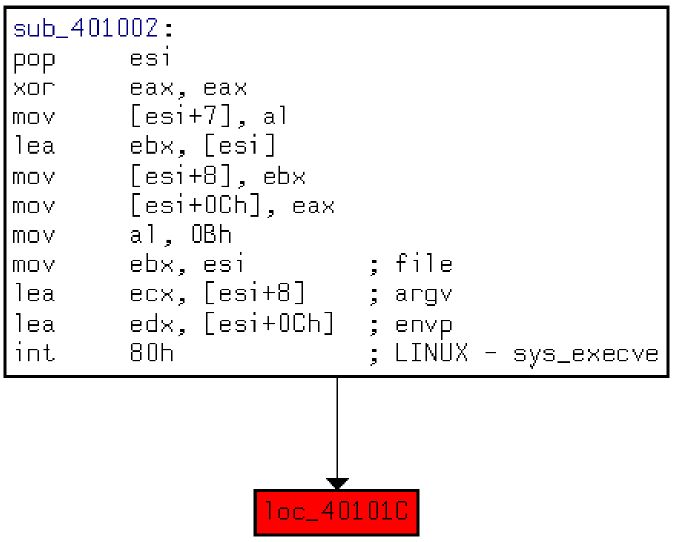


```
start:
jmp     short loc_40101C

loc_40101C:
call    sub_401002
das
bound   ebp, [ecx+6Eh]
das
jnb     short near ptr dword_401090

00401028:
and     eax, [ecx+41h]
inc     ecx
inc     ecx
inc     edx
inc     edx
inc     edx
inc     edx

dword_401090
byte_401031
```



```
sub_401002:
pop     esi
xor     eax, eax
mov     [esi+7], al
lea     ebx, [esi]
mov     [esi+8], ebx
mov     [esi+0Ch], eax
mov     al, 0Bh
mov     ebx, esi        ; file
lea     ecx, [esi+8]    ; argv
lea     edx, [esi+0Ch]  ; envp
int     80h             ; LINUX - sys_execve

loc_40101C
```

Figure 3: Back to our shellcode.

Other than the simple branches we have already looked at, conditionals are also a means to create loops and more complex code structures.

Onto loops

Loops are of course a crucial component of any code. This feature allows for iteration. This is the repeated execution of a block of code until a defined condition has been met. Each occurrence of running the code block being looped is an iteration of that code block.

There are two primary means to creating a loop in shellcode. These are:

1. Through the use of conditional jumps (as we have covered in prior articles), or
2. The use of a LOOPx instruction.

The LOOPx instructions are defined by the condition code that sets how the loop will branch. The main loops in assembly include:

• loop	Loop where ECX does not equal zero (and usually for short jumps),
• loopz	Loop if the register ECX is equal to zero,
• loope	Loop if the register ECX is equals a value it is compared to,
• loopnz	Loop if the register ECX is not equal to zero, or
• loopne	Loop if the register ECX does not equal a value it is compared to.

The “loop” instruction (without a condition) is generally used for small jumps where the branch is located less than 128bytes from the start of the loop. On each iteration of a LOOPx instruction, the system will subtract one from the ECX register.

What forms a loop?

All shellcode loops are composed of five (5) parts. These are:

1. A control variable. Each loop will contain a set of variables that can be evaluated to see if the loop should continue or end.
2. The initial value that initialises the loop has to be set for each of the control variables.
3. The block of code that acts as the body of the loop. This is the code that is run at each iteration of the loop.
4. The modification process. This stage changes the control variable.
5. An end condition. Although not strictly necessary (it is possible to have an endless loop) it is generally considered necessary to have some end to the loop such that it stops and does not run eternally.

Loops are an important component of creating shellcode. They enable the author to obscure their code (through encryption and decryption routines), to add port and IP scanning functions into the shellcode, to enact denial of services attacks and to create keystroke loggers amongst other things. Although there are many forms of looping instructions, the primary ones we will address are “*for loops*” and “while loops”.

For Loops

We can see a simple for loop disassembled into machine code in Figure 4.

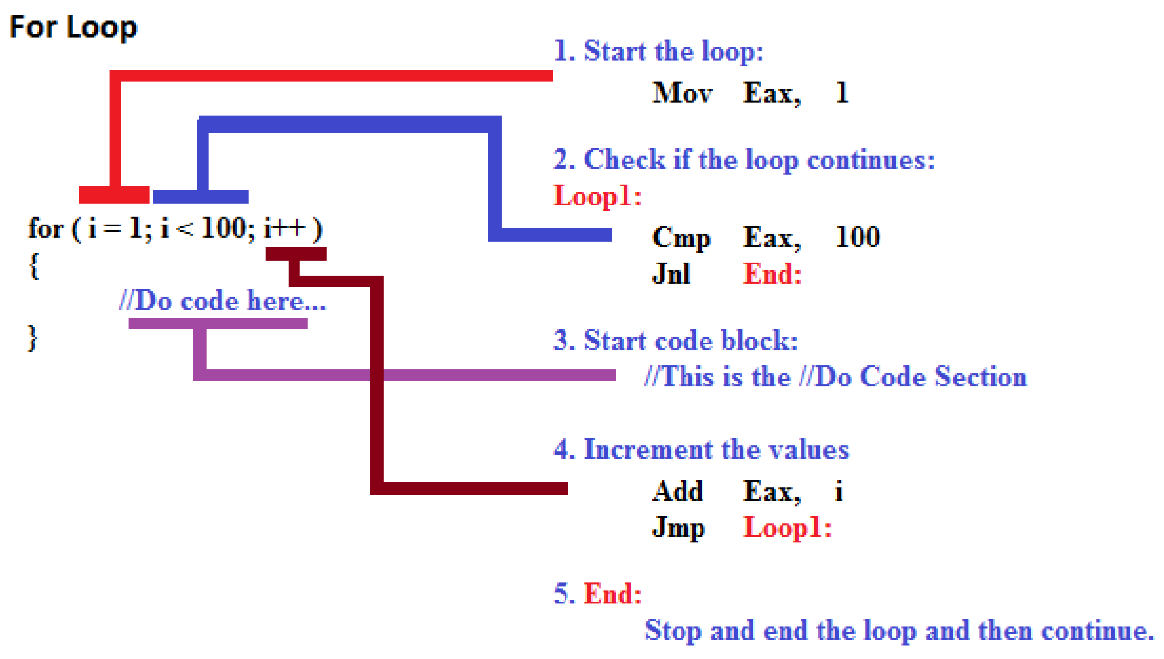


Figure 4: A For Loop in action

All loops are actually functionally equivalent (Zakharov, 1999) and can be written in different ways. We define them as we do for reasons of elegance and performance, an art more than a science.

In the “For loop” the initialisation, update routine and ending conditions are specified at the start of the loop. This is the primary difference to a while loop where the ending conditions are defined at the end of the loop and the control and update routine is set within the body of the loop. There are of course many ways to represent even a simple for loop and this makes the reversing process far more complex than it may seem it should be (and hence also comes to why there are as yet no truly automated decompilers).

From this, we can quickly deduce that a stopping condition at the start of the loop would best fit a “For Loop” whilst a stopping condition located at the end of the loop best forms a “While Loop”.

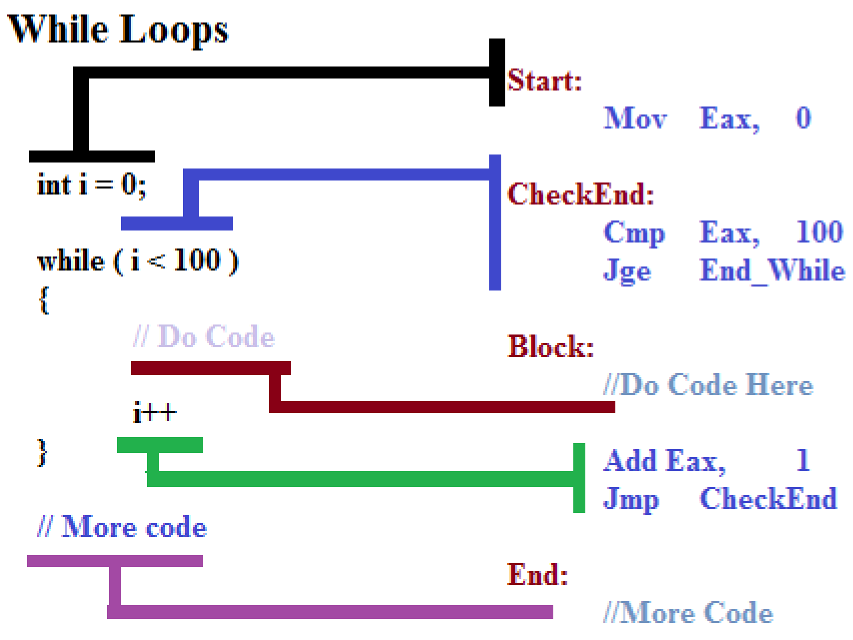
In the example (Figure 4), we start by setting our variable “i” to a value of 0 and create a routine to increment this value by one on each iteration of the routine. The loop is set to end or complete when the value of “i” reaches or exceeds 100. This means that our loop will iterate 100 times.

The C/C++ code is listed to the left of the figure with the functionally equivalent assembly code listed on the right. In order to initialise our variable “i” in assembly, we have set the EAX register to contain the value 0. As this i8s a “For Loop”, the completion or ending condition is checked at the start of the loop and we have this written as a “CMP EAX, 100” assembly instruction where the conditional jump (JNL) is taken if EAX is greater than or equal to 100. Basically, we loop until the value stored in EAX equals 100.

The value in the EAX register is incremente4d by 1 each time the code block is iterated and the check routine at (2) is again engaged.

While Loops

In the following example (Figure 5) we can see the distinction on how a “While Loop” is generally created in assembly.



```
int i = 0;

while ( i < 100 )
{
        // Do Code

        i++
}
// More code
```

```
Start:
        Mov    Eax,    0
CheckEnd:
        Cmp    Eax,    100
        Jge    End_While
Block:
        //Do Code Here

        Add Eax,       1
        Jmp    CheckEnd
End:
        //More Code
```

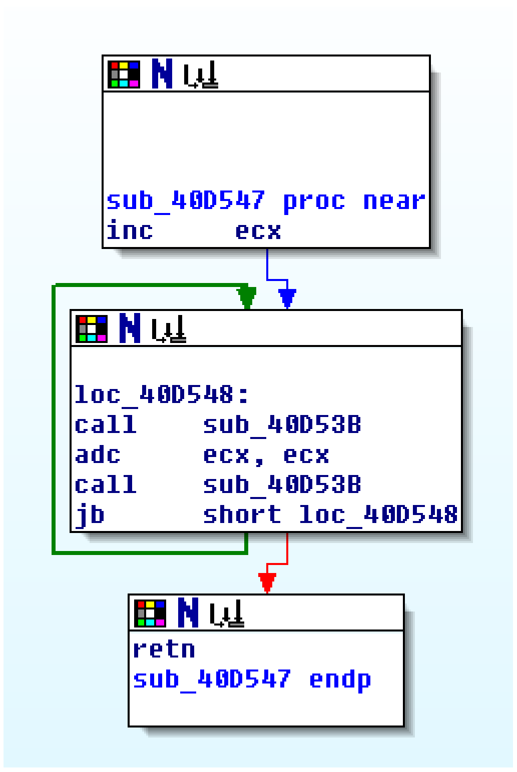
Figure 5: A While loop in action

Here we again start with initialising the variable “i” to zero. In this case, this is equivalent to setting the EAX register to hold a value of 0. Next, the loop routine evaluates the check. This is it compares “i” to the value we have set for our iterations (100) to see if it is larger or equal. In assembly, we have a CMP instruction to evaluate the EAX register against the value 100. The code block is run if and only if the value held in EAX is less than 100. When the code has run, the value held in EAX is incremented by 1 (or the value “i” in the higher level code).

In the two examples we have provided, the “while loop” and the “for loop”, the assembly code we have provided is functionally equivalent. The code could be rearranged to provide the same results.

A real life loop

If we take a look at a function from the NSPack compression routine (Wright, 2010) we can see a simple loop from a real life packing routine (figure 6).



```
sub_40D547 proc near
inc     ecx

loc_40D548:
call    sub_40D53B
adc     ecx, ecx
call    sub_40D53B
jb      short loc_40D548

retn
sub_40D547 endp
```

Figure 6:   A loop from a routine in NSPack.

In this example, a JB (or Jump if (unsigned) below with the carry flag set or CF = 1) is used to see if the block iterates.

A simpler loop

We also have another means to creating loops in shellcode, the LOOPx instruction set. These instructions decrement ECX and jump to the memory location set in the command or not depending on whether a specified condition is set.

The form of the instruction is: Loop   <memory location>

In this instruction set, the

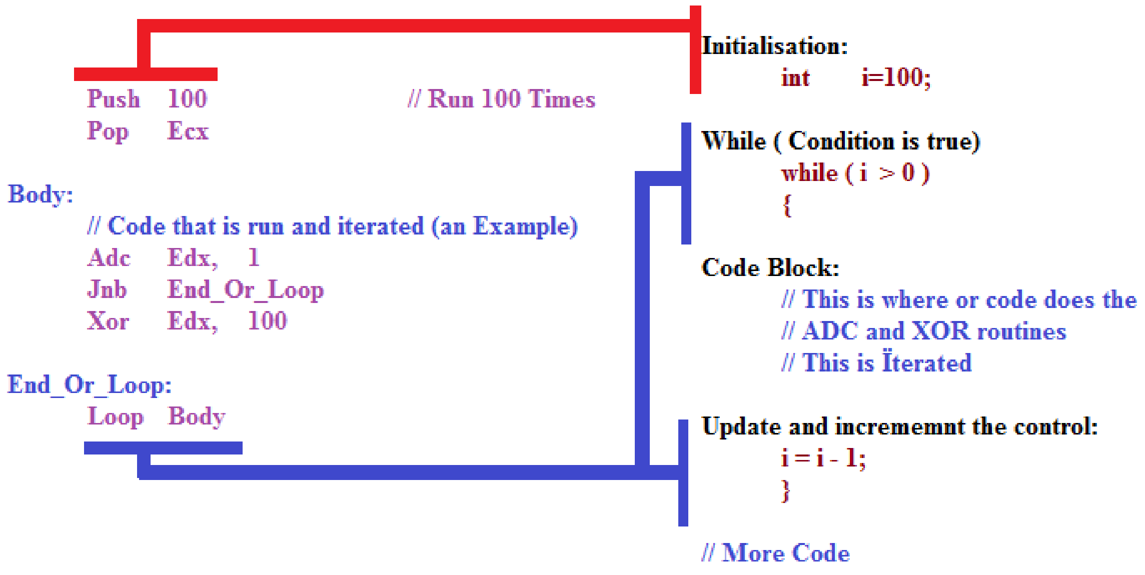
- loop	unless decrementing ECX caused its value to become zero.
- loope	loop if equal
- loopz	loop if zero
- loopne	loop if not equal
- loopnz	loop if not zero

The difference between the instructions “loop” and “loopz” are in the conditions. The instruction “loop” awaits the value in the ECX register becoming zero whereas the “loopz” instruction has a condition based on the zero flag being set when the ECX register has been decremented.

For instance in the following instruction the loop would return to location 0x4558820 if the value in ECX was not equal to zero.

```
loop     0x4558820
```

We can see how this works in figure 7 with a higher level representation as a “while loop”.



```
        Push    100                 // Run 100 Times
        Pop     Ecx

Body:
        // Code that is run and iterated (an Example)
        Adc     Edx,    1
        Jnb     End_Or_Loop
        Xor     Edx,    100

End_Or_Loop:
        Loop    Body
```

```
Initialisation:
        int     i=100;

While ( Condition is true)
        while ( i > 0 )
        {

Code Block:
        // This is where or code does the
        // ADC and XOR routines
        // This is Ïterated

Update and incrememnt the control:
        i = i - 1;
        }

// More Code
```

*Figure 7:	A loop using LOOPx.*

In this example, we have pushed the value 100 onto the stack and popped it back onto the ECX register. In this way we have set ECX as a counter from 100 down to zero. The memory location specified by “**Body:**” is the start of the loop and when we arrive at the “**Loop Body**” instruction the code either returns to the previous code block and iterates this again if the value held in ECX first decremented by 1 and then checked to see if it is greater than zero. If the value stored in ECX has been decremented to zero, the loop ends and the code continues linearly. In our example, the code is iterated (the loop is run in other words) one hundred times.

Loops are frequently utilised in order to make shellcode more difficult to detect. Using shifts for instance, the code can be encrypted or obscured making it more difficult to analyse and for IDS/IPS systems to detect and stop it.

Conclusion

At this point we have learnt the basics of assembly code and the means to set conditions and loops. We will follow this up with functions and calls before moving to the actual hooking process in coming articles. When we then put all of this together, we will have the foundations for creating shellcode for exploits and hence an understanding of the process that penetration testers and hackers use in exploiting systems. With these skills, you will see how it is possible to either create your own exploit code from scratch or even to modify existing exploit code to either add functionality or in order to bypass signature based IDS/IPS filters.

With this knowledge, you will learn just how easy it is for sophisticated attackers to create code that can bypass many security tools. More, armed with this knowledge you will have the ability to reverse engineer attack code and even malware allowing you to determine what the attacker was intending to launch against your system. In this way, you can improve your forensic and incident response skills.

In learning shellcode, we gain a deep knowledge and appreciation of the systems we are managing and attempting to secure. This process does take time and practice, but it is well worth the effort.

Through this process, we will not only learn how to successfully modify the shellcode we have copied from others, extending its use, but to also learn to create our own. More, we will be able to reverse engineer hostile shellcode and to understand what purposes it has been created for. In this, we see just how difficult it is to stop attacks.

As we have noted in prior articles, shellcode can be said to have a shelf life. As samples become popular and are used more widely in the underground community, they are slowly added into IDS and Anti-Malware signatures. Widely deployed shellcode, including that used in the Metasploit project, has a particularly low shelf-life. This is not to say that it will not be useful against many sites, but that it will be less likely to have value in testing highly secure sites.

In many cases, the alteration of small sections of the shellcode can result in the signatures that have been created to detect, alert and block it becoming ineffective. For instance, in the last article we learnt that making small changes to a piece of existing shellcode to run "*/bin/csh*" in place of the standard call to "*/bin/sh*" can increase the useful life of the shellcode. As we start to learn how shellcode is created and formed, we can also start to alter it and extend it running different payloads or changing its form to avoid detection. In this article, we have now learnt to extend our skills into changing loops. With a simple change from a For Loop to a While Loop, we are often evading many signature based controls.

Author's bio

About the Author:

Dr Craig Wright is a lecturer and researcher at Charles Sturt University and executive vice – president (strategy) of CSCSS (Centre for Strategic Cyberspace+ Security Science) with a focus on collaborating government bodies in securing cyber systems. With over 20 years of IT related experience, he is a sought-after public speaker both locally and internationally, training Australian and international government departments in Cyber Warfare and Cyber Defence, while also presenting his latest research findings at academic conferences.

In addition to his security engagements Craig continues to author IT security related articles and books. Dr Wright holds the following industry certifications, GSE CISSP, CISA, CISM, CCE, GCFA, GLEG, GREM and GSPA. He has numerous degrees in various fields including *a Master's degree in Statistics, and a Master's Degree in Law specialising in International Commercial Law. Craig is working on his second doctorate, a PhD on the Quantification of Information Systems Risk.*

References

Foster, J., Osipov, V., Bhalla, N., & Heinen, N. (Eds.). (2005). *Buffer Overflow Attacks: Detect, Exploit, Prevent*: Syngress, USA.

Wright, C. S. (2010). Packer Analysis Report-Debugging and unpacking the NsPack 3.4 and 3.7 packer. *Sans Reading Room*, from http://www.sans.org/reading_room/whitepapers/malicious/packer-analysis-report-debugging-unpacking-nspack-34-37-packer_33428

Zakharov, V. A. (1999). On the decidability of the equivalence problem for orthogonal sequential programs. *Grammars, 2*(3), 271-281.

Zillion. (2002). Writing Shellcode, from http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html