USER-STACK:

essential knowledge to Memory Corruption study

Filipe Xavier de Oliveira
engfilipeoliveira89@gmail.com
filipe.xavier@tempest.com.br

ABSTRACT

Over the years, attacks on memory corruption have become complex and distant from the reality of many security analysts and researchers. User-Stack is one of the primary topics surrounding memory corruption techniques; even so, it has often been poorly studied. Besides, being interested in memory corruption, and not mastering the knowledge about User-Stack, will certainly bring frustrations to those who wish to pursue a career in the field. Therefore, this article aims to teach, within the scope of the Windows operating system, not only the principles, but above all, the defense and attack aspects related to the User-Stack; aiming to serve both those who want to start their knowledge on the topic, as well as those who already have some experience with the stack.

Keywords: Stack, Canary, Convention, Thread.

INTRODUCTION

Stack-related attacks have had their moment of glory in the past. When it wasn’t too complicated to successfully exploit vulnerable software, this could easily be done through a stack overflow.

Over time, processor and operating system development teams have contributed to better security, usability and stack management. Thus, today, it’s necessary for security professionals to fully understand the systems under attack, in order to have a full understanding of the stack's operation, as well as the strategies developed to protect it.

SCOPE

Most concepts and codes to be presented, will consider the x86 architecture.
However, at the end of the article, the main differences of this architecture will be presented in comparison with the x64.

THREADS

> *“A thread is an entity within a process that Windows schedules for execution. Without it, the process’s program can’t run.”*
>
> MICROSOFT, Windows Internals Part1.

To understand the process of creating and handling the stack, we must take a step back and understand a little about the basic use of threads in Windows. A thread has some components that define it, including user-stack and kernel-stack.

The thread's stacks are part of a set – the thread’s Context. It’s important to know that there are other elements in the thread's Context. In addition, this set of information, brought by the context of the thread, is different for each architecture, in order to maintain compatibility with the different systems. Let's see below, the function responsible for creating the thread's Context of a process:

```
BOOL GetThreadContext(
  HANDLE    hThread,
  LPCONTEXT lpContext
);
```

The lpContext parameter is a pointer to the structure called CONTEXT, which contains the necessary information for the thread. Next, we have the CreateThread API, a function used to create threads:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  __drv_aliasesMem LPVOID lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

The **dwStackSize** parameter sets the stack size. If a null (zero) value is passed in the **dwStackSize** parameter, the stack will have its default size, which is 1MB. It is also possible to change the stack size using the compilation flag **/STACK:reserve** present in Microsoft C/C++. When a new process is created, Windows always establishes, by default, the **dwStackSize** to null.

Below, we have another option used in the creation of threads, where the **dwStackSize** parameter also sets the stack size:

```
LPVOID CreateFiber(
  SIZE_T                dwStackSize,
  LPFIBER_START_ROUTINE lpStartAddress,
  LPVOID                    lpParameter
);
```

A thread has two types of main structures at the level of the operating system: ETHREAD and KTHREAD. Whereas, the KTHREAD structure is contained as the first member of ETHREAD. Through a debugger, it’s possible to obtain the following ETHREAD information[1]:

```
lkd> dt nt!_ethread
   +0x000 Tcb                : _KTHREAD
   +0x600 CreateTime         : _LARGE_INTEGER
   +0x608 ExitTime           : _LARGE_INTEGER
   +0x608 KeyedWaitChain     : _LIST_ENTRY
   +0x618 PostBlockList      : _LIST_ENTRY
   +0x618 ForwardLinkShadow : Ptr64 Void
   +0x620 StartAddress       : Ptr64 Void
   +0x628 TerminationPort    : Ptr64 _TERMINATION_PORT
   +0x628 ReaperLink         : Ptr64 _ETHREAD
   +0x628 KeyedWaitValue     : Ptr64 Void
   +0x630 ActiveTimerListLock : Uint8B
```

Below, we can see the KTHREAD, with the elements destined for the stack in bold:

```
lkd> dt nt!_kthread
   +0x000 Header             : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget      : Uint8B
   +0x028 InitialStack       : Ptr64 Void
   +0x030 StackLimit         : Ptr64 Void
   +0x038 StackBase          : Ptr64 Void
   +0x040 ThreadLock         : Uint8B
   +0x048 CycleTime          : Uint8B
   +0x050 CurrentRunTime     : Uint4B
   +0x054 ExpectedRunTime    : Uint4B
   +0x058 KernelStack        : Ptr64 Void
   +0x060 StateSaveArea      : Ptr64 _XSAVE_FORMAT
   +0x068 SchedulingGroup    : Ptr64 _KSCHEDULING_GROUP
   +0x070 WaitRegister       : _KWAIT_STATUS_REGISTER
   +0x071 Running            : UChar
```

[1] For the following examples, the Windbg debugger in Kernel mode was used on the local machine.

```
   +0x072 Alerted           : [2] UChar
   +0x074 AutoBoostActive   : Pos 0, 1 Bit
   +0x074 ReadyTransition   : Pos 1, 1 Bit
   +0x074 WaitNext          : Pos 2, 1 Bit
   +0x074 SystemAffinityActive : Pos 3, 1 Bit
   +0x074 Alertable         : Pos 4, 1 Bit
   +0x074 UserStackWalkActive : Pos 5, 1 Bit
```

For User-Stack learning, it isn’t necessary to understand in depth the elements of the ETHREAD structure. The important thing here, is to visualize the link that the stacks have with the thread.

Another structure that also carries information about the stack is the TEB (Thread Environment Block). However, unlike the previous ones, this one needs to exist in the memory addressing of the process.

In the case below, using the **!teb** command, it’s possible to view the TEB of a process, with the elements destined for the stack also in bold:

```
0:000> !teb
TEB at 005fc000
    ExceptionList:        0036f904
    StackBase:            00370000
    StackLimit:           0036c000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 005fc000
    EnvironmentPointer:   00000000
    ClientId:             00001900 . 0000022c
    RpcHandle:            00000000
    Tls Storage:          00875a88
    PEB Address:          005f9000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0
```

By performing attach in a process and executing the "~" command, it’s possible to view all threads in the process:

```
0:001> ~
   0  Id: 1ae8.1e60 Suspend: 1 Teb: 0000005b`50329000 Unfrozen
.  1  Id: 1ae8.9cc Suspend: 1 Teb: 0000005b`5033b000 Unfrozen
```

It’s possible to notice the existence of two threads, 0 and 1. With the **~nk** command, it’s possible to view the stack of each thread, where "**n**" will be the thread's identifier.

Notice that the **~1k** command shows the stack of thread number 1:

```
0:001> ~1k
 # Child-SP          RetAddr               Call Site
00 0000005b`501ffc68 00007ffa`eb4cd3cb     ntdll!DbgBreakPoint
01 0000005b`501ffc70 00007ffa`eaac7c24
ntdll!DbgUiRemoteBreakin+0x4b
02 0000005b`501ffca0 00007ffa`eb46cea1
KERNEL32!BaseThreadInitThunk+0x14
03 0000005b`501ffcd0 00000000`00000000
ntdll!RtlUserThreadStart+0x21
```

Stacks are crucial for the functioning of the threads, and without both, it’s impossible for a program to remain active in an operating system.

THE FIRST THREAD OF A PROCESS

Whenever we open a program, a new process is created. During the creation of each new process, a thread is also made. Whenever a thread is executed, a new space in the memory is reserved so that both local variables, as well as parameters and return addresses of function calls are allocated. In addition, new frames are created and inserted into memory whenever a thread makes a new call. A frame is basically the collection of all the data needed to perform a function. It’s precisely this reserve of memory that we call stack.

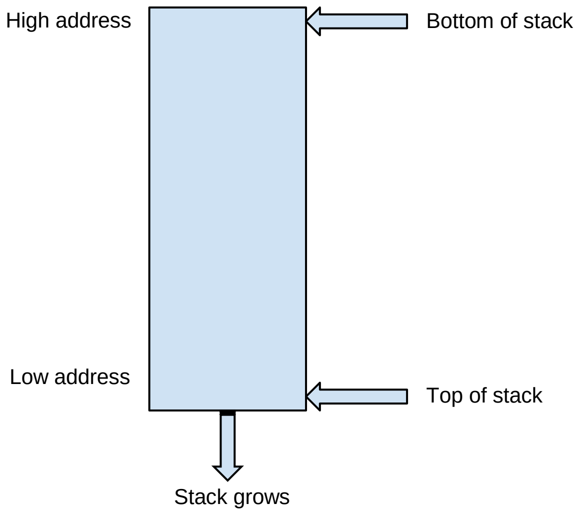
The Windows memory manager handles 3 types of stacks: The dpc stack, the user stack and the kernel stack. We remind you that the Windows memory manager will automatically reserve 1 MB of memory for the User Stack when creating the new thread.

As the name implies, the best way to explain how the stack works is by comparing it to a “stack of plates”. Where the elements to be removed (POP) or inserted (PUSH) will always be at the top of the pile; that's why the stack is known for its LIFO (Last In, First Out) semantics.

The image below illustrates a layout about the stack structure during a function call in an x86 architecture.

Function Parameter 1
Function Parameter 2
Function Parameter N
Return Address
Frame Pointer
Exception Handler Frame
Local Variable 1
Local Variable 2
Local Variable N
Function Saved Registers

Next, we have a drawing showing how the stack growth develops. We illustrate that it starts from a memory address, growing downwards towards lower addresses:



For that reason, when we talk about the top of the stack in x86, we talk about the low address. However, many debuggers have an inverted view of the stack.

For a better understanding of how the stack works, we can load an executable for testing, such as Windows **notepad.exe**, which will be our example.

Using the command **x notepad!*main***, you can view the main of the program:

```
0:000> x notepad!*main*
00cdf8da          notepad!__mainCRTStartup (void)
00ce3440          notepad!_imp____getmainargs = <no type
information>
00ccc303          notepad!WinMain (_WinMain@16)
00cdf8d0          notepad!WinMainCRTStartup (_WinMainCRTStartup)
```

To insert a breakpoint in the **notepad!WinMainCRTStartup** function, we use the **bu** command. To list the inserted breakpoints, we use the **bl** command, as can be seen below:

```
0:000> bu notepad!WinMainCRTStartup
0:000> bl
     0 e Disable Clear  00cdf8d0     0001 (0001)  0:****
notepad!WinMainCRTStartup
```

So, when executing the program, we stop at the first function of our main:

```
0:000> g
ModLoad: 75cc0000 75ce5000   C:\Windows\SysWOW64\IMM32.DLL
Breakpoint 0 hit
eax=0019bf1a ebx=005b2000 ecx=00cdf8d0 edx=04040010 esi=00cdf8d0
edi=00cdf8d0
eip=00cdf8d0 esp=0063f930 ebp=0063f93c iopl=0         nv up ei pl
zr na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000247
notepad!WinMainCRTStartup:
00cdf8d0 e8b0080000      call    notepad!__security_init_cookie
(00ce0185)
```

Once in the **call notepad!__security_init_cookie** function, you can see the values of the records:

```
0:000> t
eax=0019bf1a ebx=005b2000 ecx=00cdf8d0 edx=04040010 esi=00cdf8d0
edi=00cdf8d0
eip=00ce0185 esp=0063f92c ebp=0063f93c iopl=0         nv up ei pl
zr na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000247
notepad!__security_init_cookie:
00ce0185 8bff            mov     edi,edi
```

Then, you can view the ESP and EBP records, on lines 01 and 02, respectively:

```
0:000> k
 # ChildEBP RetAddr  
00 0063f928 00cdf8d5     notepad!__security_init_cookie
01 0063f92c 75656359     notepad!WinMainCRTStartup+0x5
02 0063f93c 772d7c24     KERNEL32!BaseThreadInitThunk+0x19
03 0063f998 772d7bf4     ntdll!__RtlUserThreadStart+0x2f
```

```
04 0063f9a8 00000000     ntdll!_RtlUserThreadStart+0x1b
```

From the image above, it can be seen that the values contained in the stack are instructions written by the process in order to create the first thread to execute the program. In addition, it’s possible to observe the moment when the EIP (Extended Instruction Pointer) is stopped over the MOV EDI, EDI instruction. It’s also possible to notice that the EBP (Extended Base Pointer) register is pointing to the BaseThreadInitThunk API of **kernel32.dll**. For this reason, this is one of the first frames to be created, given the initialization of the first thread that accompanies the creation of the process.

Each time a thread makes a function call, a new frame is inserted into the stack before the program executes any function; as a result, the operating system executes a series of calls that are part of the process of creating the threads themselves.

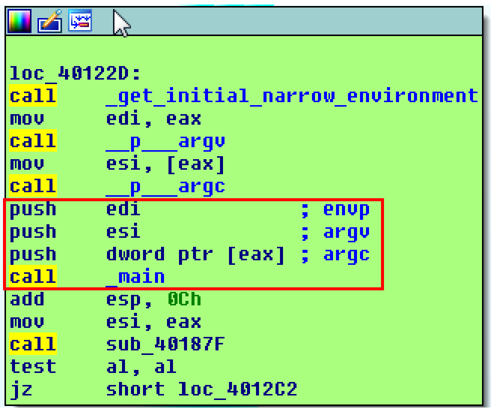
Therefore, it’s important to note that, for each language or version of the compiler, the thread initiation functions may be different in their aspect, but not in their result. It’s important to realize that, in most cases, the first functions to be loaded into a program will be related to the creation of its first thread.

PROLOGUE, CODE AND EPILOGUE

As we saw earlier, a program's first function does not always belong to its code. See the following code example:

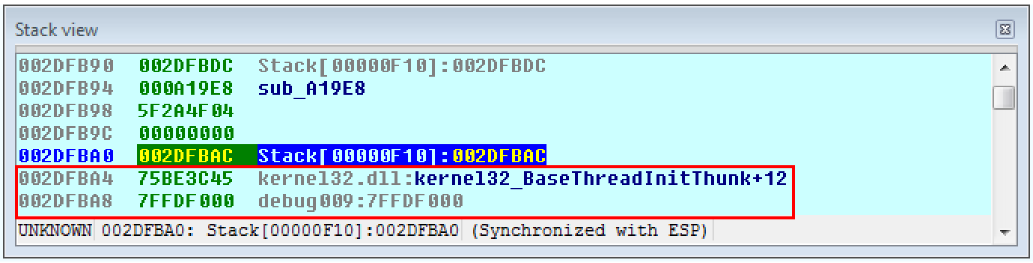
```
int main(int argc, char* argv[])
{
    char buffer[20];

    printf("Type anything\n");
    gets_s(buffer, sizeof(buffer));

}
```

The image below shows the Entry Point of the program and the arguments that belong to the Main of the program:



```
loc_40122D:
call    _get_initial_narrow_environment
mov     edi, eax
call    __p___argv
mov     esi, [eax]
call    __p___argc
push    edi             ; envp
push    esi             ; argv
push    dword ptr [eax] ; argc
call    _main
add     esp, 0Ch
mov     esi, eax
call    sub_40187F
test    al, al
jz      short loc_4012C2
```

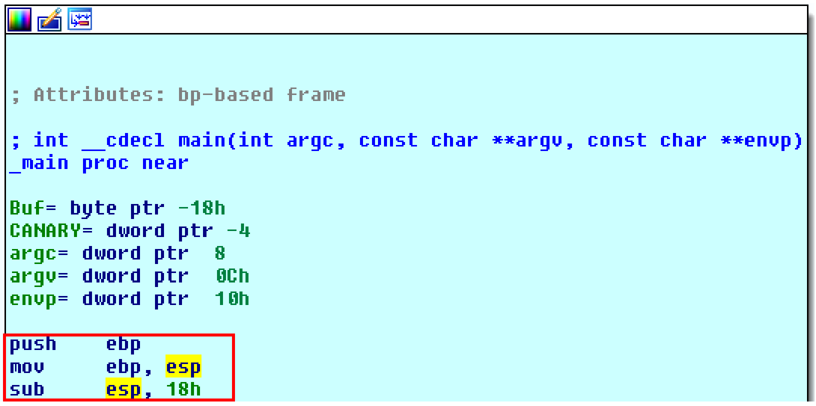
In the next image, it's possible to see the stack at the moment when the EIP is pointed at our **call _main**. Note that below the address 0x002DFBA0 is the pointer to BaseThreadInitThunk, as seen earlier:



Stack view

```
002DFB90  002DFBDC  Stack[00000F10]:002DFBDC
002DFB94  000A19E8  sub_A19E8
002DFB98  5F2A4F04
002DFB9C  00000000
002DFBA0  002DFBAC  Stack[00000F10]:002DFBAC
002DFBA4  75BE3C45  kernel32.dll:kernel32_BaseThreadInitThunk+12
002DFBA8  7FFDF000  debug009:7FFDF000
UNKNOWN 002DFBA0: Stack[00000F10]:002DFBA0 (Synchronized with ESP)
```

Once inside the function, you can view the following set of instructions:

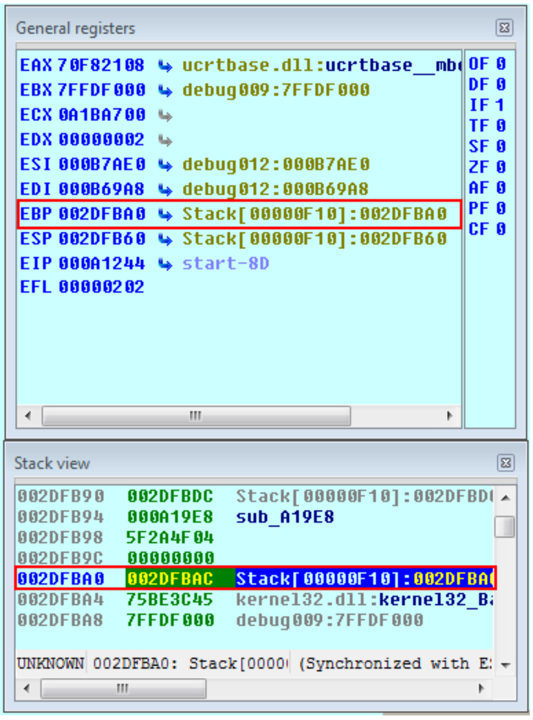


```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

Buf= byte ptr -18h
CANARY= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 18h
```

```
.text:000A1040 push    ebp
.text:000A1041 mov     ebp, esp
.text:000A1043 sub     esp, 18h
```

The 3 instructions above form the prologue of a function. The prologue of a function ensures that the stack is prepared for the new function to run smoothly.

Before explaining what each of the lines above means, we need to understand the purpose of each register. The EBP register points to an address in memory, usually being the base of the frame, in order to facilitate the location of the arguments and variables of a function, see the image below:

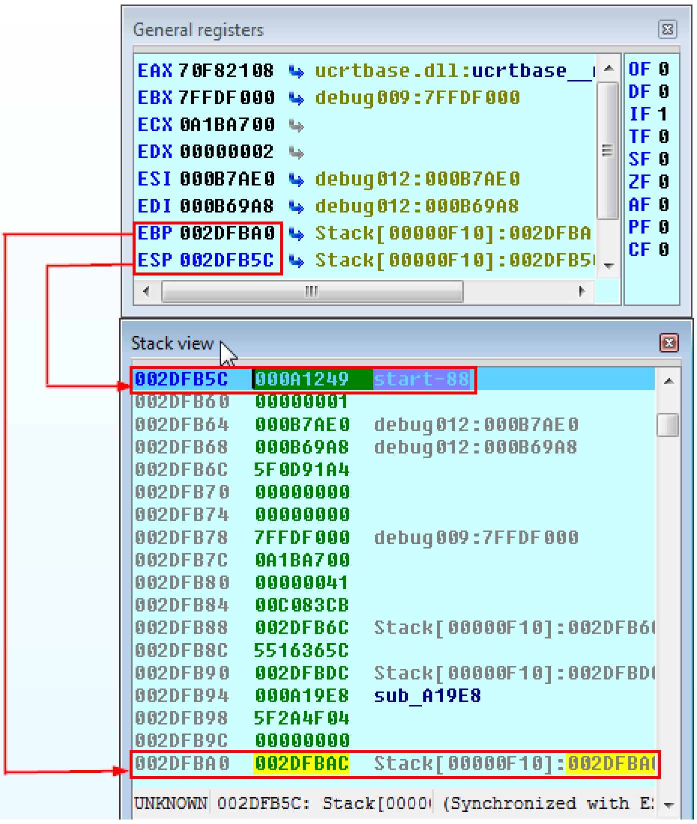


```
EAX 70F82108 ↳ ucrtbase.dll:ucrtbase__mb
EBX 7FFDF000 ↳ debug009:7FFDF000
ECX 0A1BA700 ↳
EDX 00000002 ↳
ESI 000B7AE0 ↳ debug012:000B7AE0
EDI 000B69A8 ↳ debug012:000B69A8
EBP 002DFBA0 ↳ Stack[00000F10]:002DFBA0
ESP 002DFB60 ↳ Stack[00000F10]:002DFB60
EIP 000A1244 ↳ start-8D
EFL 00000202
```

```
OF 0
DF 0
IF 1
TF 0
SF 0
ZF 0
AF 0
PF 0
CF 0
```

```
002DFB90  002DFBDC  Stack[00000F10]:002DFBD
002DFB94  000A19E8  sub_A19E8
002DFB98  5F2A4F04
002DFB9C  00000000
002DFBA0  002DFBAC  Stack[00000F10]:002DFBA
002DFBA4  75BE3C45  kernel32.dll:kernel32_B
002DFBA8  7FFDF000  debug009:7FFDF000

UNKNOWN 002DFBA0: Stack[0000 (Synchronized with E
```

The ESP register, on the other hand, is the index that will always point to the top of the stack, as shown in the following image:



```
EAX 70F82108 ↳ ucrtbase.dll:ucrtbase__
EBX 7FFDF000 ↳ debug009:7FFDF000
ECX 0A1BA700 ↳
EDX 00000002 ↳
ESI 000B7AE0 ↳ debug012:000B7AE0
EDI 000B69A8 ↳ debug012:000B69A8
EBP 002DFBA0 ↳ Stack[00000F10]:002DFBA
ESP 002DFB5C ↳ Stack[00000F10]:002DFB5

OF 0
DF 0
IF 1
TF 0
SF 0
ZF 0
AF 0
PF 0
CF 0
```

```
002DFB5C  000A1249  start-88
002DFB60  00000001
002DFB64  000B7AE0  debug012:000B7AE0
002DFB68  000B69A8  debug012:000B69A8
002DFB6C  5F0D91A4
002DFB70  00000000
002DFB74  00000000
002DFB78  7FFDF000  debug009:7FFDF000
002DFB7C  0A1BA700
002DFB80  00000041
002DFB84  00C083CB
002DFB88  002DFB6C  Stack[00000F10]:002DFB6
002DFB8C  5516365C
002DFB90  002DFBDC  Stack[00000F10]:002DFBD
002DFB94  000A19E8  sub_A19E8
002DFB98  5F2A4F04
002DFB9C  00000000
002DFBA0  002DFBAC  Stack[00000F10]:002DFBA

UNKNOWN 002DFB5C: Stack[0000 (Synchronized with E:
```

Returning to the aforementioned set of instructions, note that the second instruction (PUSH EBP) inserts the EBP value into the stack via PUSH. The second instruction (MOV EBP, ESP) moves the ESP register to EBP, because if the ESP register is lost, it will still be possible to restore it via EBP, thus establishing the start of a new stack frame. Later on, we will see how the EBP register can be used.

The table below shows the state of the stack at the time we went through the prologue:

0x002DFB58 002DFBA0 Stack[00000F10]:002DFBA0	PUSH EBP
0x002DFB5C 000A1249 start-88	RETURN ADDRESS

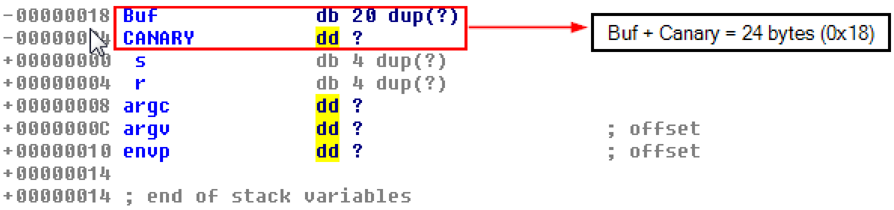
The return address guarantees the location of the program to be returned to when exiting the function; because, when a function call is executed, the function's return address is inserted (PUSH) in the stack.

It’s also important to realize that the stack grows from top to bottom, decreasing the stack pointer (ESP). Note that, in the table above, the stack address is 0x002DFB5C.

Thus, when the PUSH EBP is executed, the ESP value is decreased by 4 bytes (0x002DFB58).

The last instruction (SUB ESP, 0x18) subtracts 18 bytes or 24 in decimal from the stack pointer. This subtraction is important, as it creates space for local variables.

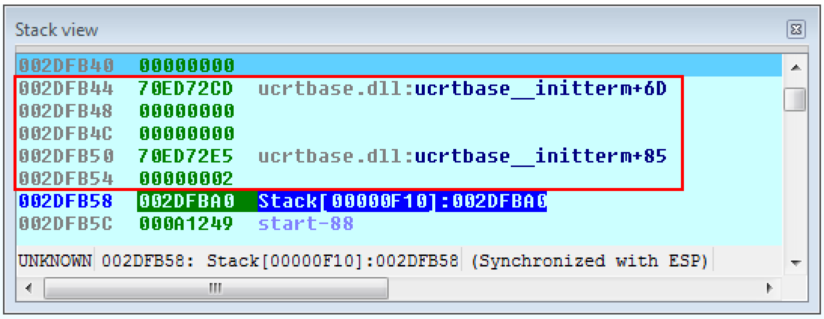
Looking at the disassembler, it is easy to see which local variables are reserved:



```
-00000018 Buf             db 20 dup(?)
-00000004 CANARY          dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 argc            dd ?
+0000000C argv            dd ?                    ; offset
+00000010 envp            dd ?                    ; offset
+00000014
+00000014 ; end of stack variables
```

Buf + Canary = 24 bytes (0x18)

When we subtract 24 bytes from the stack pointer, a space is reserved on the stack. The figure below shows the stack after subtraction:



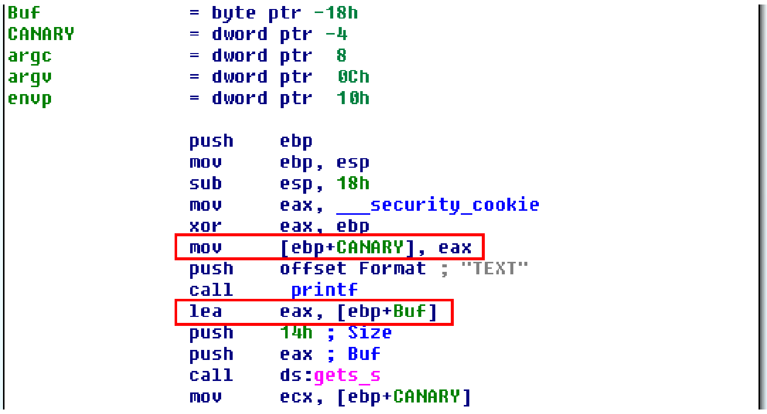
```
002DFB40  00000000
002DFB44  70ED72CD  ucrtbase.dll:ucrtbase__initterm+6D
002DFB48  00000000
002DFB4C  00000000
002DFB50  70ED72E5  ucrtbase.dll:ucrtbase__initterm+85
002DFB54  00000002
002DFB58  002DFBA0  Stack[00000F10]:002DFBA0
002DFB5C  000A1249  start-88
UNKNOWN 002DFB58: Stack[00000F10]:002DFB58 (Synchronized with ESP)
```

The new address of the stack pointer directs to 0x002DFB40. Thus, when we add 0x18 to address 0x002DFB40, we'll have the value 0x002DFB58. Likewise, if we subtract 0x18 from address 0x002DFB58, we'll arrive at the value of the current stack pointer. Looking at the table below, it's possible to observe the stack when the subtraction occurs:

0x002DFB40	Stack Pointer	Sub esp, 0x18
0x002DFB58	Saved EBP	Push EBP

0x002DFB5C	Return address	Call _main

After the stack pointer (ESP) was set to receive the local variables, the next instructions related to the local variables were stored in the stack. The next image shows the relationship of the variables to the stack:



```
Buf             = byte ptr -18h
CANARY          = dword ptr -4
argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                mov     eax, ___security_cookie
                xor     eax, ebp
                mov     [ebp+CANARY], eax
                push    offset Format ; "TEXT"
                call     printf
                lea     eax, [ebp+Buf]
                push    14h ; Size
                push    eax ; Buf
                call    ds:gets_s
                mov     ecx, [ebp+CANARY]
```

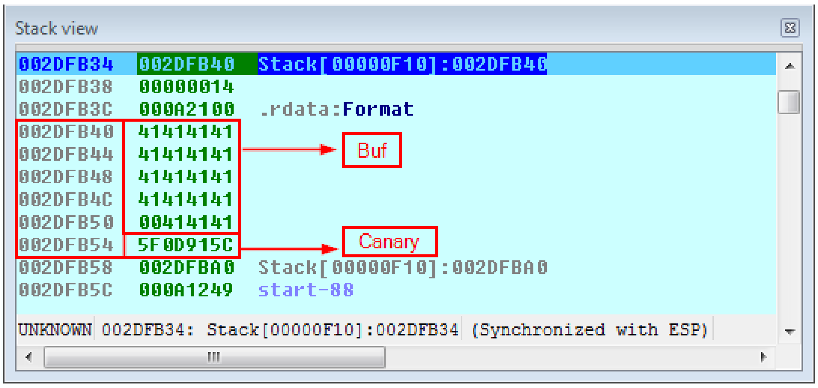
The CANARY and Buf variables are stored in the stack using the following instructions:

```
mov     [ebp+CANARY], eax
lea     eax, [ebp+Buf]
```

The EBP register is used as a pointer that references the position of each local variable. This action is, by default, the purpose of the EBP register; unlike the ESP register, which changes continuously during the execution of the function. For that reason, it would be difficult to use ESP as the base of the frame.

As we saw earlier, the program has two local variables, Buf and Canary. Later on, we'll talk about the variable Canary, as it's a variable inserted by the compiler in the program routine.

Now, let's see how the stack behaves when receiving the two variables with their respective values:



```
002DFB34  002DFB40  Stack[00000F10]:002DFB40
002DFB38  00000014
002DFB3C  000A2100  .rdata:Format
002DFB40  41414141
002DFB44  41414141
002DFB48  41414141
002DFB4C  41414141
002DFB50  00414141
002DFB54  5F0D915C
002DFB58  002DFBA0  Stack[00000F10]:002DFBA0
002DFB5C  000A1249  start-88
UNKNOWN 002DFB34: Stack[00000F10]:002DFB34 (Synchronized with ESP)
```

The table below shows, in more detail, the stack when it receives the variables:

0X002DFB34	Buf pointer = 002DFB40	push eax
0X002DFB38	Buf size = 0x14	push 14h
0X002DFB3C	Push 'TEXT' string	push offset Format ; "TEXT"
0X002DFB40	Buf = 41414141	lea eax, [ebp+Buf]
0X002DFB44	Buf = 41414141	
0X002DFB48	Buf = 41414141	
0X002DFB4C	Buf = 41414141	
0X002DFB50	Buf = 00414141	
0X002DFB54	Canary	mov [ebp+CANARY], eax
0X002DFB58	Saved EBP	Push ebp
0X002DFB5C	Return Address	Call _main

After executing the body of the program, we arrive at the epilogue; whose responsibility is to ensure that the stack is restored to the point before the **CALL _main** call. The image below shows the epilogue of the function:

```
.text:000A1069 add     esp, 0Ch
.text:000A1075 mov     esp, ebp
.text:000A1077 pop     ebp
```

The first instruction adds 0x0C to the stack pointer. The purpose of this sum is to restore the stack after the three pushes, which are the parameters used in the function, namely: Buf, Canary and push offset Format (printf). It’s important to remember that, with each new function, it’s necessary to restore the stack to its previous state.

To find out if the value added in ESP is correct, just add the values in bytes of each parameter in our routine:

```
Buf (4 bytes) + Canary (4 bytes) + Push offset Format(4 bytes) =
0XCh
```

After adding 0xC to the stack pointer, the new ESP value will be the address 0X002DFB40:

0X002DFB40	Buf = 41414141	lea eax, [ebp+Buf]
0X002DFB44	Buf = 41414141	
0X002DFB48	Buf = 41414141	
0X002DFB4C	Buf = 41414141	
0X002DFB50	Buf = 00414141	
0X002DFB54	Canary	mov [ebp+CANARY], eax
0X002DFB58	Saved EBP	Push ebp
0X002DFB5C	Return Address	Call _main

The second instruction moves the value from EBP to ESP, remembering that EBP has the same value that ESP had when stored in the prologue. The following table shows the moment when ESP will point to the bottom of the frame:

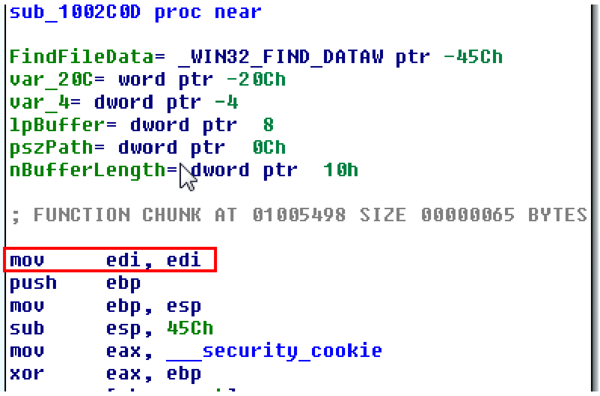
0X002DFB58	Saved EBP	mov esp, ebp
0X002DFB5C	Return Address	pop ebp

Keeping in mind that, at this moment, the value of ESP is equal to that of EBP, the POP EBP instruction will leave the ESP register pointing to the return address. In addition, through these techniques, the process is able to reserve the necessary space for the functions, as well as for their return.

HOT PATCHING

Hot Patching is a method of updating a system or program without having to download a new version. Updates are made dynamically to the executable or the vulnerable system.

After explaining the three usual instructions used in the prologue, we will present a fourth instruction, which can often be found. That instruction is MOV EDI, EDI. Below is an example of a prologue with it:



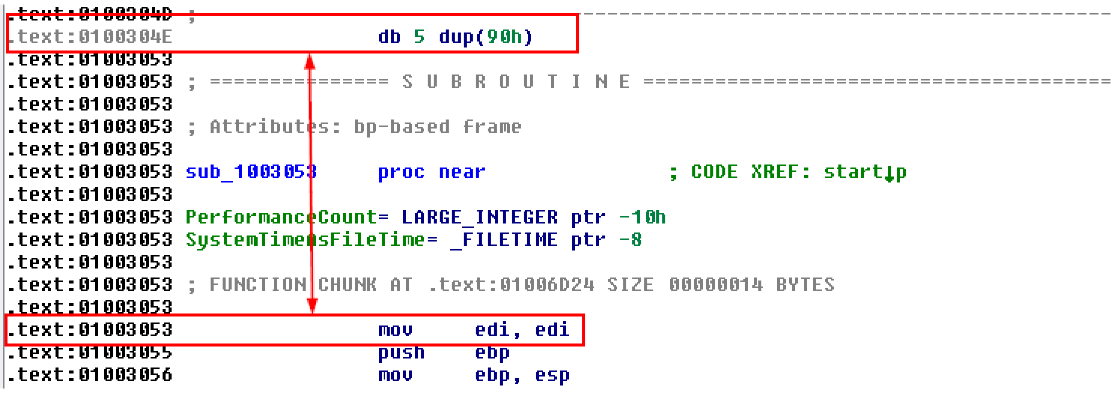
```
sub_1002C0D proc near

FindFileData= _WIN32_FIND_DATAW ptr -45Ch
var_20C= word ptr -20Ch
var_4= dword ptr -4
lpBuffer= dword ptr  8
pszPath= dword ptr  0Ch
nBufferLength= dword ptr  10h

; FUNCTION CHUNK AT 01005498 SIZE 00000065 BYTES

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 45Ch
mov     eax, ___security_cookie
xor     eax, ebp
```

Would this instruction be a part of the prologue? How can an instruction with null effect have any practical use? The MOV EDI, EDI instruction has no meaning, because even the flags aren’t changed when it is executed. However, to further complicate its concept, Microsoft defines it as one that "is equivalent to two NOP (NO OPERATION), and that has enough space for a short jump (**jmp short**)".

Well, the instruction MOV EDI, EDI does not belong to the prologue of the function; and its usefulness is to leave a space reserved for future modifications, which is fast enough to be worth its existence in the program.

Once the instruction is replaced by a jmp short (2 bytes), we can be taken to a space that precedes the first instruction of the entry point of that routine, whose region has 5 bytes reserved for writing. However, unlike the short jump, the 5-byte 'full jump' is able to redirect the program flow to any new written instructions for hot patching.

The image below shows the 5 bytes that precede our entry point:



```
.text:0100304D ; ------------------------------------------------------------
.text:0100304E                 db 5 dup(90h)
.text:01003053
.text:01003053 ; =============== S U B R O U T I N E =======================================
.text:01003053
.text:01003053 ; Attributes: bp-based frame
.text:01003053
.text:01003053 sub_1003053     proc near               ; CODE XREF: start↓p
.text:01003053
.text:01003053 PerformanceCount= LARGE_INTEGER ptr -10h
.text:01003053 SystemTimeAsFileTime= _FILETIME ptr -8
.text:01003053
.text:01003053 ; FUNCTION CHUNK AT .text:01006D24 SIZE 00000014 BYTES
.text:01003053
.text:01003053                 mov     edi, edi
.text:01003055                 push    ebp
.text:01003056                 mov     ebp, esp
```

If there was no MOV EDI, EDI instruction, the hot patching would have to be performed on an instruction that has an impact on the program, or that is difficult to restore. Which would lead to correction risks. Thus, overwriting an instruction that is important or difficult to reconstruct would not be a good option. For this reason, Microsoft has decided to insert one more instruction, the modification cost of which would be zero, as well as the risks of altering important parts of the program. Besides, considering the execution time, the instruction MOV EDI, EDI is still 50% faster than the two consecutive NOPS.

It’s worth remembering that you can add the patch option on demand to Visual C++ using the compilation flag **/hotpatch**.

CALLING CONVENTIONS

When a code is compiled in Visual C/C++, calling conventions are established. In many cases, developers belittle the issue, so that most of the time, arguing about it is unnecessary.

Traditionally, functions made in C/C++ have the same behavior in relation to the stack. Identifying the different types of calling conventions teaches us different ways of how arguments can be passed and cleared from the stack.

All of our examples, so far, are being compiled using the standard __**cdecl** convention, where arguments are passed from right to left, example:

```
Push Arg1
Push Arg2
Push Arg3
Call function
Add esp, 0xC
```

Below is an example of a call:

```
int __cdecl function(int a, int b, int c);

int function(int a, int b, int c)
{
    int num = a + 2;
    int num2 = b + 3;
    int num3 = c + 4;
    int sum = num + num2 + num3;

    return sum;
}

int main()
{
    return function(10, 20, 30);
}
```

The table below shows the stack layout when using the above function:

EBP + 16	int c
EBP + 12	int b
EBP + 8	int a
EBP + 4	Return address
EBP	Saved ebp
EBP - 4	num
EBP - 8	num2
EBP - 12	num3
EBP - 16	sum [ESP]

A second convention, **__stdcall**, can be called using the ***/Gz*** compiler flag that specifies its use for all functions. The two main characteristics of **__stdcall** are: The arguments are executed from right to left, and the stack is cleaned up by the function that called it. Therefore, it is possible to create smaller executables than the **__cdecl** convention; because, with **__cdecl**, the stack cleanup must be generated for each function call.

Below is a code example with the **__stdcall** convention changed:

```
int __stdcall sum (int a, int b);

int sum (int a, int b)
{
     return a + b;
}


int main()
{
```

```
int c = sum(2, 3);

printf("%d", c);

}
```

The third convention, __**fastcall**, is made in such a way that some arguments are written in registers. One of the advantages of __**fastcall** is that operations carried out through registers are faster than the stack.

You can use the **/Gr** compilation flag to specify __**fastcall** for all functions. The main feature of __**fastcall** is that its first two function arguments are entered in the ECX and EDX registers. The rest of the arguments are inserted in the stack from right to left.

The following table informs the main differences between the three types of convention:

__cdecl	It is the standard convention, and its biggest advantage is due to the fact that it allows the existence of a variable number of arguments. Generally, executables tend to be larger.
__stdcall	It doesn’t allow a variable number of arguments, it’s possible to create smaller executables.
__fastcall	Write some of its arguments in the registers, while the rest are inserted into the stack. Its biggest advantage is the speed in the call of functions.

It’s important not to forget, that when mixing the types of calling conventions, some problems can be incurred, these are called calling conventions mismatch[2].

[2] To learn more about calling conventions mismatch we recommend reading the blog The Old New Thing [https://devblogs.microsoft.com/oldnewthing/20040115-00/?p=41043]

FRAME POINTER OMISSION

Frame Pointer Omission (FPO) is a technique that uses the base frame register (EBP) as a multi-purpose register. Generally, its use is linked to the speed gain; thus, the compiler uses EBP to store various types of data. FPO can cause problems especially for those who are programming directly in assembly; therefore, care should be taken when using the stack pointer.

Debugging a program with FPO can be confusing; because, in the event of a crash without the frame pointer, the debugger won't be able to generate the stack trace, the generation will only occur if the symbols are present. Therefore, the use of FPO makes debugging difficult.

If an executable crash using FPO, your dump won't contain the stack frame pointer. Therefore, the debugger won't be able to generate the stack trace correctly from that dump. The stack trace can only be restored completely if the symbols are present in the program. Since, the FPO information is recorded in the program's symbol file. Anyway, there's a possibility to restore that stack trace manually; however, this task will never be trivial, besides, it cannot be said that it's always possible.

The following images show the difference between a routine of an executable without FPO and another with FPO, respectively:

```
0:000> uf 006a1040
Perilogue!main

   11 006a1040 55              push    ebp
   11 006a1041 8bec            mov     ebp,esp
   11 006a1043 83ec18          sub     esp,18h
   11 006a1046 a104306a00      mov     eax,dword ptr
[Perilogue!__security_cookie (006a3004)]
   11 006a104b 33c5            xor     eax,ebp
   11 006a104d 8945fc          mov     dword ptr [ebp-4],eax
   14 006a1050 6800216a00      push    offset Perilogue!`string'
```

```
(006a2100)
   14 006a1055 e8b6ffffff      call    Perilogue!printf (006a1010)
   15 006a105a 8d45e8          lea     eax,[ebp-18h]
   15 006a105d 6a14            push    14h
   15 006a105f 50              push    eax
   15 006a1060 ff15b4206a00    call    dword ptr
[Perilogue!_imp__gets_s (006a20b4)]
   17 006a1066 8b4dfc          mov     ecx,dword ptr [ebp-4]
   17 006a1069 83c40c          add     esp,0Ch
   17 006a106c 33cd            xor     ecx,ebp
   17 006a106e 33c0            xor     eax,eax
   17 006a1070 e804000000      call
Perilogue!__security_check_cookie (006a1079)
   17 006a1075 8be5            mov     esp,ebp
   17 006a1077 5d              pop     ebp
   17 006a1078 c3              ret
```

```
0:000> uf 006b1040
Perilogue!main

   11 006b1040 83ec18          sub     esp,18h
   11 006b1043 a104306b00      mov     eax,dword ptr
[Perilogue!__security_cookie (006b3004)]
   11 006b1048 33c4            xor     eax,esp
   11 006b104a 89442414        mov     dword ptr [esp+14h],eax
   14 006b104e 6800216b00      push    offset Perilogue!`string'
(006b2100)
   14 006b1053 e8b8ffffff      call    Perilogue!printf (006b1010)
   15 006b1058 8d442404        lea     eax,[esp+4]
   15 006b105c 6a14            push    14h
   15 006b105e 50              push    eax
   15 006b105f ff15b4206b00    call    dword ptr
[Perilogue!_imp__gets_s (006b20b4)]
   17 006b1065 8b4c2420        mov     ecx,dword ptr [esp+20h]
   17 006b1069 83c40c          add     esp,0Ch
   17 006b106c 33cc            xor     ecx,esp
   17 006b106e 33c0            xor     eax,eax
   17 006b1070 e804000000      call
Perilogue!__security_check_cookie (006b1079)
   17 006b1075 83c418          add     esp,18h
   17 006b1078 c3              ret
```

Note that, in the first image, the use of the EBP register is constant. In the second image, the EBP record doesn't appear in the routine at any time. The only register used as a guide in the stack is ESP itself, when using FPO.

In the following image, it's possible to view the EBP when the application with FPO is stopped at its Entry Point:

```
0:000> k
 # ChildEBP RetAddr
00 006afb80 006b1249     Perilogue!main // [EBP]
01 (Inline) --------     Perilogue!invoke_main+0x1c
02 006afbc8 75656359     Perilogue!__scrt_common_main_seh+0xfa
03 006afbd8 772d7c24     KERNEL32!BaseThreadInitThunk+0x19
04 006afc34 772d7bf4     ntdll!__RtlUserThreadStart+0x2f
05 006afc44 00000000     ntdll!_RtlUserThreadStart+0x1b
0:000> r
eax=751b10e0 ebx=00901000 ecx=00000000 edx=00000000 esi=00a44750
edi=00a449c8
eip=006b1040 esp=006afb84 ebp=006afbc8 iopl=0         nv up ei pl
nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
Perilogue!main:
006b1040 83ec18          sub     esp,18h
```

The next image shows the first local variable on the stack; and, below, the return address of the function. Observe that, above the return address, there's no base pointer:

```
0:000> p
eax=3ad9c74f ebx=00901000 ecx=00000000 edx=00000000 esi=00a44750
edi=00a449c8
eip=006b104a esp=006afb6c ebp=006afbc8 iopl=0         nv up ei pl
nz na po nc
```

```
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000202
Perilogue!main+0xa:
006b104a 89442414        mov     dword ptr [esp+14h],eax
ss:002b:006afb80=00000002
```

```
0:000> k
 # ChildEBP RetAddr
00 006afb80 006b1249     Perilogue!main+0xe // [esp+14h]
01 (Inline) --------     Perilogue!invoke_main+0x1c // return
02 006afbc8 75656359     Perilogue!__scrt_common_main_seh+0xfa
03 006afbd8 772d7c24     KERNEL32!BaseThreadInitThunk+0x19
04 006afc34 772d7bf4     ntdll!__RtlUserThreadStart+0x2f
05 006afc44 00000000     ntdll!_RtlUserThreadStart+0x1b
```

Note that the FPO doesn't use the EBP register as we saw earlier. Caution is required when creating executables with the FPO, as reliance on a single register makes the stack more susceptible to failure[3], such that any corruption in its pointers will cause failure in the search for local variables or at the return point. It's recommended to be careful about the amount of garbage that the program can write into the stack.

STACK OVERFLOW

A stack overflow occurs when a variable stored in the stack is filled with a buffer that exceeds its size, causing arbitrary writing on the stack. Usually, the basic exploitation scenarios that involve stack overflow are those whose return addresses are changed to a controlled memory address.

The variants that cause a stack overflow are diverse, and in many cases may be the result of other vulnerabilities. For this reason, we won't analyze the technical aspects of other bug classes, focusing only on those that make direct reference to the stack.

[3] For another example of a stack failure involving FPO we recommend the following reading [https://devblogs.microsoft.com/oldnewthing/20040116-00/?p=41023]

Regardless of the bug class, what we care about is the behavior that the stack will exhibit from the exploitation of a vulnerability.

Below is an example of a stack overflow[4]:

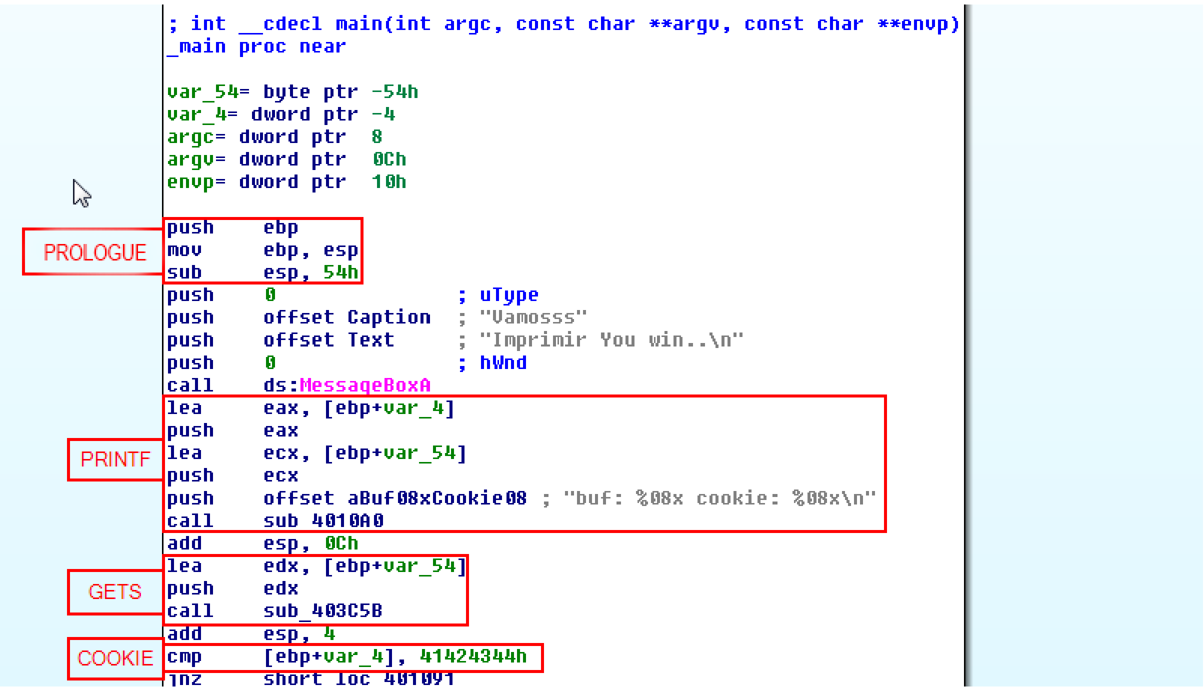
```
int main(int argc, char** argv)
{
      int cookie;
      char buf[80];

      printf("buf: %08x cookie: %08x\n", &buf, &cookie);
      gets(buf);

      if (cookie == 0x41424344)
            printf("you win!\n");
}
```

Looking at the code above, we see the use of the gets[5] function, whose defect is that it doesn't have a value that controls the size of its input.

The image below shows the main routine, without symbols:

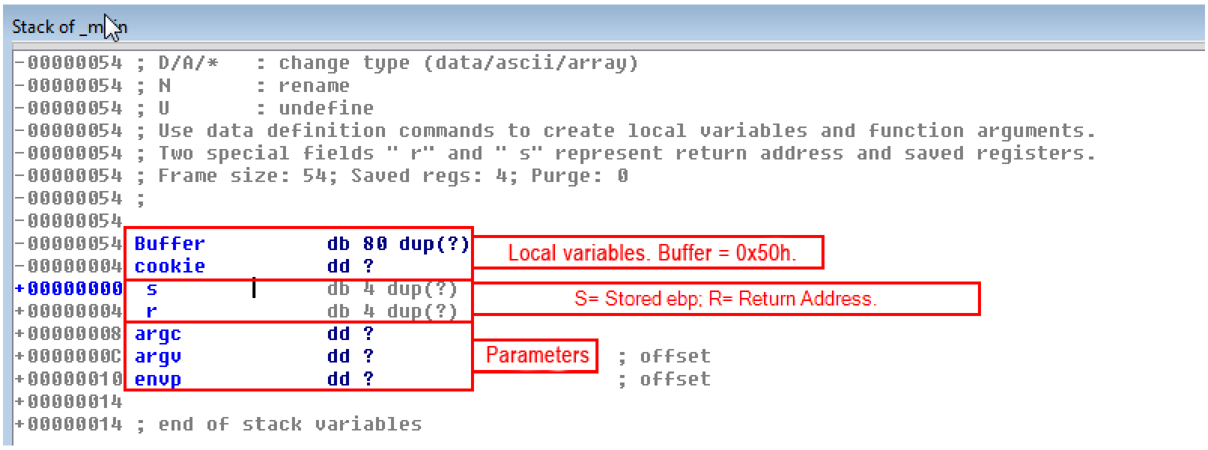
[4] The example in question was taken from [http://ricardonarvaja.info/WEB/EXPLOITING%20Y%20REVERSING%20USANDO%20HERRAMIENTAS%20FREE/EJERCICIOS/]

[5] More about the gets function can be found at [(https://docs.microsoft.com/pt-br/cpp/c-runtime-library/gets-getws?view=msvc-160)]



```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

var_54= byte ptr -54h
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 54h
push    0               ; uType
push    offset Caption  ; "Vamosss"
push    offset Text     ; "Imprimir You win..\n"
push    0               ; hWnd
call    ds:MessageBoxA
lea     eax, [ebp+var_4]
push    eax
lea     ecx, [ebp+var_54]
push    ecx
push    offset aBuf08xCookie08 ; "buf: %08x cookie: %08x\n"
call    sub_4010A0
add     esp, 0Ch
lea     edx, [ebp+var_54]
push    edx
call    sub_403C5B
add     esp, 4
cmp     [ebp+var_4], 41424344h
jnz     short loc_401091
```

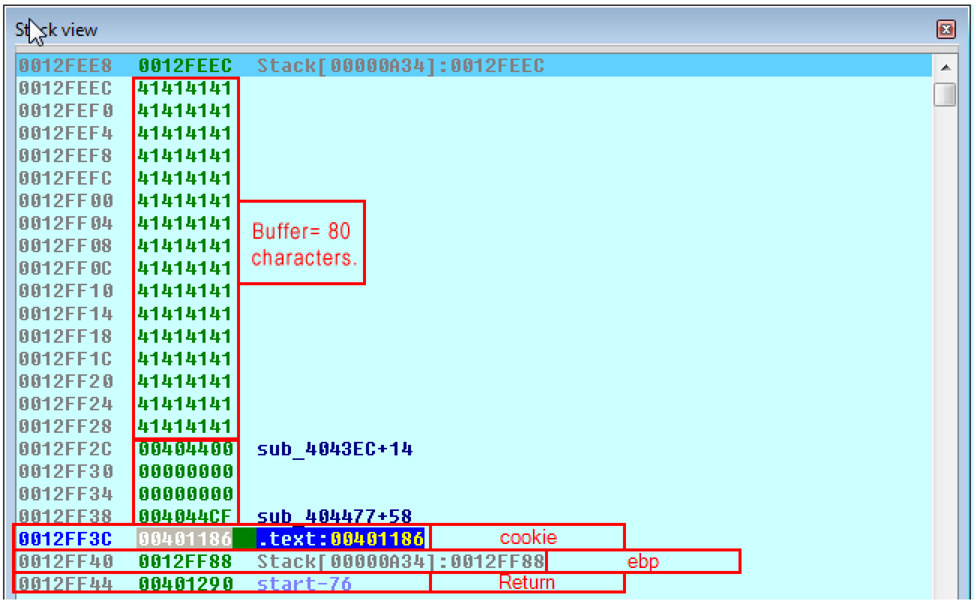
In the previous image, we see two local variables, **var_4** and **var_54** (it's possible to rename the variables with more meaningful names, which we'll do later). Now let's look at the stack of the Main function and the arrangement of the local variables, already renamed, plus the parameters:



Local variables. Buffer = 0x50h.
S= Stored ebp; R= Return Address.
Parameters

```
Stack of _main
-00000054 ; D/A/*    : change type (data/ascii/array)
-00000054 ; N        : rename
-00000054 ; U        : undefine
-00000054 ; Use data definition commands to create local variables and function arguments.
-00000054 ; Two special fields " r" and " s" represent return address and saved registers.
-00000054 ; Frame size: 54; Saved regs: 4; Purge: 0
-00000054 ;
-00000054
-00000054 Buffer          db 80 dup(?)
-00000004 cookie          dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 argc            dd ?
+0000000C argv            dd ?                  ; offset
+00000010 envp            dd ?                  ; offset
+00000014
+00000014 ; end of stack variables
```

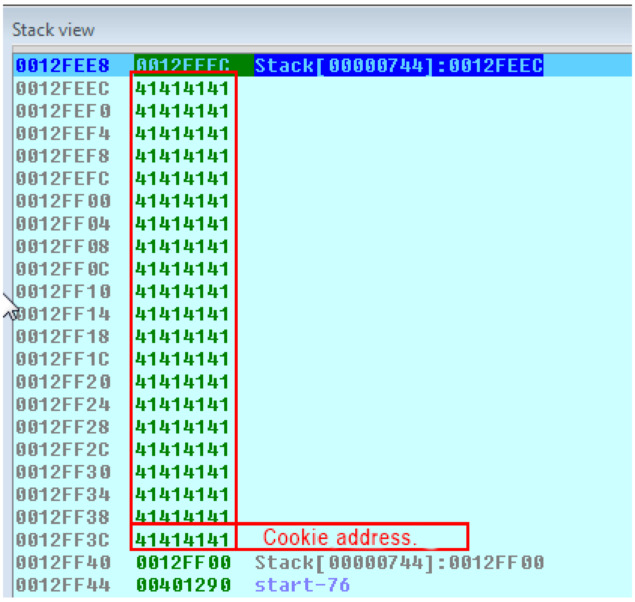
Let's fill the stack with the maximum value that the Buffer variable expects to receive (80 in decimal). The image below shows the stack filled with 80 characters, and the cookie variable:



```
Stack view
0012FEE8  0012FEEC  Stack[00000A34]:0012FEEC
0012FEEC  41414141
0012FEF0  41414141
0012FEF4  41414141
0012FEF8  41414141
0012FEFC  41414141
0012FF00  41414141
0012FF04  41414141
0012FF08  41414141
0012FF0C  41414141
0012FF10  41414141
0012FF14  41414141
0012FF18  41414141
0012FF1C  41414141
0012FF20  41414141
0012FF24  41414141
0012FF28  41414141
0012FF2C  00404400  sub_4043EC+14
0012FF30  00000000
0012FF34  00000000
0012FF38  004044CF  sub_404477+58
0012FF3C  00401186  .text:00401186
0012FF40  0012FF88  Stack[00000A34]:0012FF88
0012FF44  00401290  start-76
```

Since the gets function doesn’t limit the size of the user's input, it’s possible to insert more bytes into the stack until we reach the address of the cookie variable. Starting from the address 0x12FF2C until 0x12FF3C we have 5 DWORDS, which results in 20 bytes from the expected end of the buffer to the cookie address.

Inserting 100 bytes, we see that the value of the cookie was overwritten by 41414141:

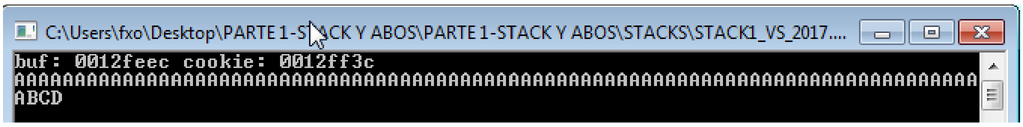


```
Stack view
0012FEE8  0012FEEC  Stack[00000744]:0012FEEC
0012FEEC  41414141
0012FEF0  41414141
0012FEF4  41414141
0012FEF8  41414141
0012FEFC  41414141
0012FF00  41414141
0012FF04  41414141
0012FF08  41414141
0012FF0C  41414141
0012FF10  41414141
0012FF14  41414141
0012FF18  41414141
0012FF1C  41414141
0012FF20  41414141
0012FF24  41414141
0012FF28  41414141
0012FF2C  41414141
0012FF30  41414141
0012FF34  41414141
0012FF38  41414141
0012FF3C  41414141
0012FF40  0012FF00  Stack[00000744]:0012FF00
0012FF44  00401290  start-76
```

After looking at the application code, we see that there’s a condition to be met:

```
if (cookie == 0x41424344)
          printf("you win!\n");
```

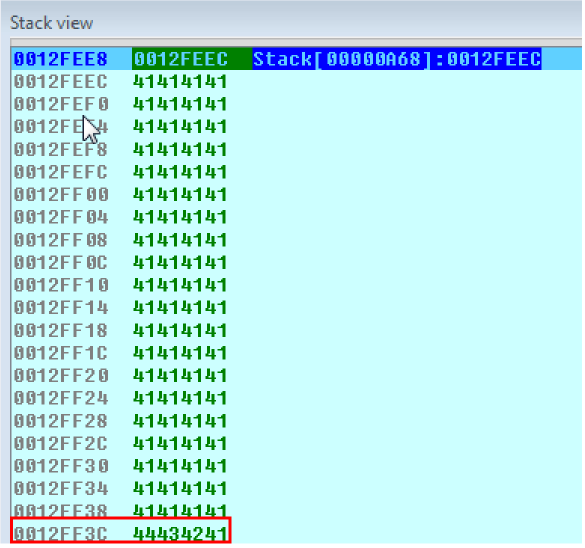
Thus, we must insert at the end of the payload the ASCII values of 0X41424344 (ABCD):



C:\Users\fxo\Desktop\PARTE 1-STACK Y ABOS\PARTE 1-STACK Y ABOS\STACKS\STACK1_VS_2017....

```
buf: 0012feec cookie: 0012ff3c
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ABCD
```

And then, finally, look at the stack:

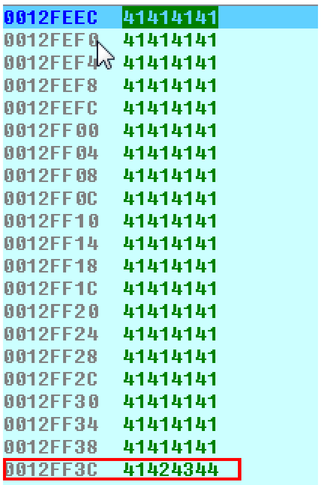


Stack view

```
0012FEE8  0012FEEC  Stack[00000A68]:0012FEEC
0012FEEC  41414141
0012FEF0  41414141
0012FEF4  41414141
0012FEF8  41414141
0012FEFC  41414141
0012FF00  41414141
0012FF04  41414141
0012FF08  41414141
0012FF0C  41414141
0012FF10  41414141
0012FF14  41414141
0012FF18  41414141
0012FF1C  41414141
0012FF20  41414141
0012FF24  41414141
0012FF28  41414141
0012FF2C  41414141
0012FF30  41414141
0012FF34  41414141
0012FF38  41414141
0012FF3C  44434241
```

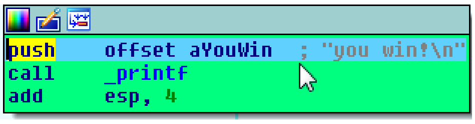
It can be seen that the comparison will never be correct, because the order of the characters is reversed in the x86 architecture. In other words, the comparison that occurs at that moment is: CMP [44434221h], 41424344h. We call this sort of ordering little endian[6]. Thus, for our comparison to be successful, we must insert the final characters in the inverted order, DCBA. Check this out in the image below:

[6] For an example of little indian see [https://searchnetworking.techtarget.com/definition/big-endian-and-little-endian#:~:text=For%20example%2C%20in%20a%20big,1000%2C%204F%20at%201001)]



```
0012FEEC  41414141
0012FEF0  41414141
0012FEF4  41414141
0012FEF8  41414141
0012FEFC  41414141
0012FF00  41414141
0012FF04  41414141
0012FF08  41414141
0012FF0C  41414141
0012FF10  41414141
0012FF14  41414141
0012FF18  41414141
0012FF1C  41414141
0012FF20  41414141
0012FF24  41414141
0012FF28  41414141
0012FF2C  41414141
0012FF30  41414141
0012FF34  41414141
0012FF38  41414141
0012FF3C  41424344
```

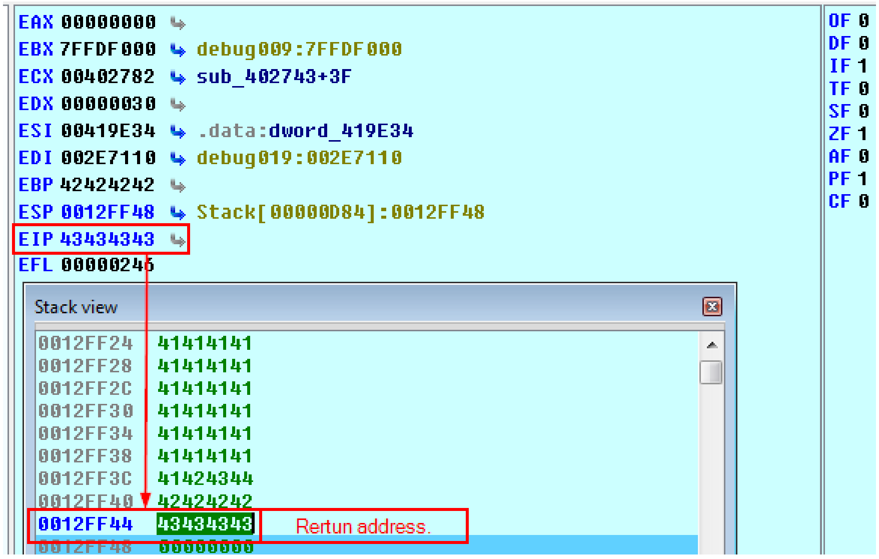
So, it was possible to redirect the flow of the program through a stack overflow:



```
push    offset aYouWin  ; "you win!\n"
call    _printf
add     esp, 4
```

In the same way that we changed the value of the cookie, it would also be possible to continue with our overflow until we overwrite the return address.

The following image will show that, continuing the overflow, it's possible to reach the return address. Note that, at this point, the EIP has the same value as the return address:



```
EAX 00000000 ↳                                   OF 0
EBX 7FFDF000 ↳ debug009:7FFDF000                 DF 0
ECX 00402782 ↳ sub_402743+3F                     IF 1
EDX 00000030 ↳                                   TF 0
ESI 00419E34 ↳ .data:dword_419E34                SF 0
EDI 002E7110 ↳ debug019:002E7110                 ZF 1
EBP 42424242 ↳                                   AF 0
ESP 0012FF48 ↳ Stack[00000D84]:0012FF48          PF 1
EIP 43434343 ↳                                   CF 0
EFL 00000246

Stack view
0012FF24  41414141
0012FF28  41414141
0012FF2C  41414141
0012FF30  41414141
0012FF34  41414141
0012FF38  41414141
0012FF3C  41424344
0012FF40  42424242
0012FF44  43434343   Rertun address.
0012FF48  00000000
```

Let's look at a second example of stack-overflow:

```
void copy(char* input)
{
    char buf[64];
    strcpy(buf, input);
}

int main(int argc, char* argv[])
{
    copy(argv[1]);
    return 0;
}
```

The second argument that the program will receive is **argv[1]**, whose value will be used as a parameter of the copy function. The copy function, on the other hand, contains both the local variable (buf) and the **strcpy** function. The latter is responsible for copying the content of the input parameter to the buf variable. The **strcpy** function is an insecure function, as it doesn’t check whether the destination buffer will support the entered value. Therefore, it’s possible to see that the code in the previous image shows a programming failure.

Let's see the situation of the stack after the execution of the program:

```
0014F710  0014F72C  Stack[00000550]:0014F72C // Argument address
buf
0014F714  0014F72C  Stack[00000550]:0014F72C // Argument address
input
0014F718  41414141 // Start of variable buf
0014F71C  41414141
0014F720  41414141
0014F724  41414141
0014F728  41414141
0014F72C  41414141
0014F730  41414141
0014F734  41414141
0014F738  41414141
0014F73C  41414141
```

```
0014F740  41414141
0014F744  41414141
0014F748  41414141
0014F74C  41414141
0014F750  41414141
0014F754  00414141 // End of variable buf
0014F758  0014F7A0  Stack[00000550]:0014F7A0 // EBP saved in the
prologue
0014F75C  00851234  start-88 // Return address
```

The first point to be controlled by our buffer would be the saved EBP value (0x0014F7A0). Due to the fact that the EBP contains the frame pointer, if the attacker controls the memory value 0x0014F7A0, the execution of the code would be transferred to the memory region of the pointer in question.

However, if the EBP cannot be controlled, we proceed with the execution by overwriting the value of the return address; that way, the attacker could still skip execution to any point controlled by him.

Once the return address is overwritten, the attacker can also begin to explore the argument following the return address.

Finally, we conclude that the concept of stack overflow is quite simple. For this reason, the important thing is to know how to exploit this type of failure, since we saw that it was possible to conduct the flow of the program, as well as change its return address. Once a stack overflow occurs, it can be exploited in a variety of ways, from replacing local variables, value of parameter pointers, return addresses, and exception structures[7], to attacks against VTable, or even against an array index that doesn't have a defined limit.

[7] Demonstration of exploitation of the exception structure through a stack overflow [https://resources.infosecinstitute.com/topic/seh-exploit/]

FLAGS DA STACK

We saw earlier how the stack can be filled easily through an overrun buffer, and that the results of an overrun won't always result in a crash. In order to maintain the integrity of the stack, the canary was created, which in Microsoft compilers is known as the guard stack, whose flag used in the compiler is ***/GS***.

The canary technique consists of inserting a check value between the local variables and the return address. The advantages are great, since the computational cost is minimal, requiring little performance to execute it.

The table below shows the stack using the canary:

Function Parameter N
Function Parameter 2
Function Parameter 1
Return Address
Frame Pointer
Canary
Exception Handler Frame
Local Variable 1
Local Variable 2
Local Variable N
Function Saved Registers

By default, Visual C++ already inserts the ***/GS*** flag at the time of compilation. Let's see in practice how the canary behaves in an application:

```
.data:00403004 ___security_cookie dd 0BB40E64Eh
```

```
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+CANARY], eax
```

O

The __**security_cookie** is generated by CRT (C RunTime) during startup, being different with each new execution. If the application does not use CRT, a call must be made to __**security_cookie**. Then, the value of the __**security_cookie** will be inserted in EAX and an XOR will be performed with the EBP value. The EBP value will also be random with each new run. In addition, the canary value will be stored in a position (DWORD) above the return address.

Using the last code example, compiled with **/GS**, we can see that the canary is located between the local variable and the frame pointer:

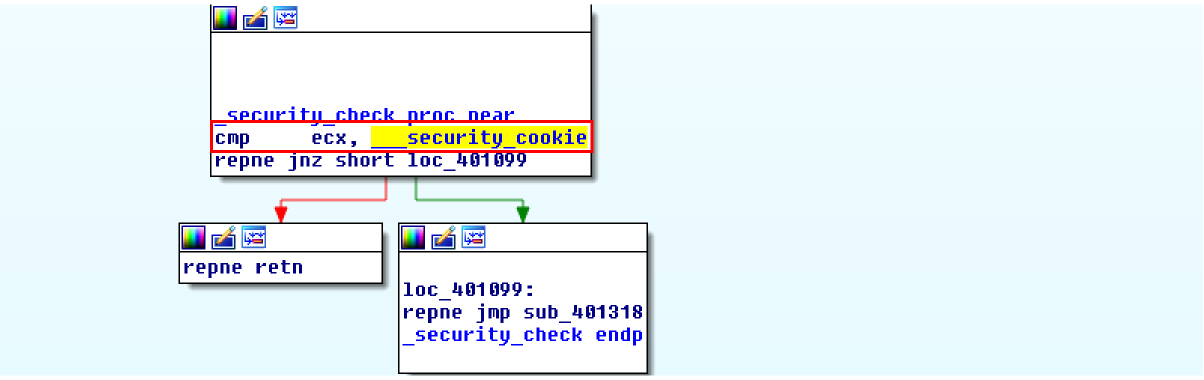
```
-00000044 Buffer          db 64 dup(?)
-00000004 CANARY          dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
```

In the routine that precedes the return address, it's possible to view the code responsible for checking the integrity of the canary value:

```
mov     ecx, [ebp+CANARY]
xor     ecx, ebp
xor     eax, eax
call    _security_check
```

When performing an XOR between the value of the canary and the original value of the EBP, it's possible to restore the value of the __**security_cookie**. In this case, since this value is the same as the one created previously, the variable CANARY hasn't been overwritten, which guarantees that the return address has also not been changed. However, in case the value is not the same as the original _**security_cookie**, the program will simply end.

The following image compares ECX with __**security_cookie**:



```
_security_check proc near
cmp     ecx, ___security_cookie
repne jnz short loc_401099
```

```
repne retn
```

```
loc_401099:
repne jmp sub_401318
_security_check endp
```

With the use of the canary, the return address, the exception handler address of a function and the function parameters will be protected. However, the canary won't prevent buffer overrun on the stack and other types of attacks.

Another important flag is **/NX**, known as DEP[8] (Data Execution Prevention). It prevents certain memory regions from being executable, so even if an attacker were able to write malicious data to the stack, he wouldn't execute them, as the memory region of the stack would be protected against code execution[9].

A third flag, also important, is **/RTC** (RunTimeChecks). The RTC performs some essential checks. For example:

- **/RTCs**: Checks the stack frame for errors at runtime. Each time a function is called, it initializes all local variables with random values, in order to avoid values from previous calls.
- **/RTCc**: Provides protection against data loss. An example of such a loss would be a casting of the ULONG type for a BYTE, where possibly data will be lost. This check, on the compiler, will show an error message whenever a cast results in data loss.

[8] Starting with Windows 7, DEP is already activated by default.

[9] However, there is a method to bypass DEP, for more details see. [https://fluidattacks.com/blog/bypassing-dep/]

- **/RTCu**: Provides protection for uninitialized variables. The compiler will always show an error whenever a variable is accessed, before its initialization.

The **/RTC** compilation flag was designed to work with builds in debug mode. Therefore, the checks carried out by the RTC don't work in programs in the release mode.

SHADOW STACK

Intel, starting in 2016, implemented in some of its chipsets, what they called CET (Control-Flow Enforcement Technology). A technology whose purpose is to protect users from control-flow hijacking attacks. However, only Windows 10 supports this technology.

The techniques implemented by CET are: Shadow Stack and Indirect branch tracking. But as this document is about the stack, we'll only cover the implementation of the Shadow Stack. Microsoft decided to call this technique, hardware-enforced stack protection; but some security researchers also refer to it as Return Flow Guard (RFG).

When CET is active, a new register is used, the SSP (Shadow Stack Pointer). The SSP register cannot be used for the same purposes as the original stack.

Like the ESP register, which points to the top of the stack, the SSP register points to the top of the shadow stack.

When the shadow stack is active, and the program is close to executing a function, at that moment, the return address is inserted in both stacks. If the return address is not the same in both stacks, the processor will generate an exception (INT 21 - Control Protection Fault).

The shadow stack technique is nothing more than a backup of the return address for each function. If the return address is overwritten, at the end of the routine, a comparison with the shadow stack will be performed to compare the return values. If the values are not the same, the program will be terminated.

After being discontinued by Microsoft, the Return Flow Guard (RFG[10]) was relaunched this year. However, this type of implementation can only be seen on intel's 11th generation processors, called Tiger Lake[11].

WHAT CHANGES IN x64?

In this section, we will cover the main differences between the x86 and x64 architecture stacks.

In the debugger, 64-bit values are represented by two 32-bit numbers, and sometimes separated by a grave accent (`). For example, 0x60000000`00000000. This value is equivalent to 0x60000000 on x86 processors.

Registers have also been extended on the x64 architecture. They now begin with the letter "**r**" instead of the letter "**e**". The x86 mnemonics are still maintained. For example, EBX still exists and is equivalent to the least significant 32 bits in the RBX register. In addition, another 8 registers were added, going from register **r8** to register **r15**.

Next, we can see the registers in an x64 architecture:

```
0:000> r
rax=0000000000000000 rbx=0000000000000000 rcx=00000000774d99fa
rdx=0000000000000000 rsi=00000000001cf520 rdi=00000000774774f0
rip=0000000077516bb0 rsp=00000000001cef30 rbp=0000000077477560
 r8=00000000001cef28  r9=0000000077477560 r10=0000000000000000
r11=0000000000000246 r12=00000000775a2c90 r13=0000000000000000
r14=00000000775a2e50 r15=000007fffffd4000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000246
```

[10] For more information [https://windows-internals.com/cet-on-windows/]
[11] Also [https://en.wikipedia.org/wiki/Tiger_Lake_(microprocessor)]

On Windows x64, we have only one type of convention:

```
rcx: Contains the first parameter passed to the function.
rdx: Contains the second parameter passed to the function.
r8:  Contains the third parameter passed to the function.
r9:  Contains the fourth parameter passed to the function.
rax: Contains the result of a function.
```

If a function has more than 4 parameters, they'll be stored in the stack from right to left. The rightmost parameter will always be stored first in the stack.

The table below shows the layout of a 64-bit stack:

Parameters for the stack
R9
R8
RDX
RCX
Return

CONCLUSION

Throughout this article, we saw that the stack is aligned and organized through the prologue, code and epilogue. In addition, we also studied how different types of calling conventions can influence our understanding of the stack. We learned how the frame point omission is able to handle the stack only with the ESP register. Finally, we saw the compilation flags that involve stack security, such as the canary and the shadow stack. Also, due to the difference between the architectures, we presented the changes of the x64 version.

We conclude by remembering that the ways in which the stacks are built and maintained – through prologues and epilogues based on different architectures or even through compilation flags – may have their peculiarities or differences; but in the end, what an attacker will always try to do is overwrite parameters, local variables, or return addresses contained in the stack, no matter how the information got there.