USER-STACK:

conhecimento essencial ao estudo de Memory Corruption

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

RESUMO

Com o passar dos anos, os ataques relacionados à *memory corruption* tornaram-se complexos e se distanciaram da realidade de muitos analistas e pesquisadores de segurança. A *User-Stack* é um dos tópicos primários que circundam as técnicas de *memory corruption*; mesmo assim, ele tem sido, muitas vezes, mal estudado. Além disso, interessar-se por *memory corruption*, e não dominar o conhecimento sobre *User-Stack*, certamente trará frustrações aos que desejam seguir carreira na área. Deste modo, este artigo tem o objetivo de ensinar, no âmbito do sistema operacional Windows, não somente os princípios, mas sobretudo, os aspectos de defesa e ataque relacionados à *User-Stac*k; visando, com isto, servir tanto aos que querem iniciar seus conhecimentos no tema, quanto aos que já possuem alguma experiência com a *stack*.

Palavras-Chave: Stack, Canary, Convention, Thread.

INTRODUÇÃO

Ataques relacionados à *stack* já tiveram seu momento de glória no passado. Quando não era muito complicado realizar uma exploração bem-sucedida num software vulnerável, o que poderia ser feito, facilmente, através de um *stack overflow*.

Com o tempo, as equipes de desenvolvimento de processadores e de sistemas operacionais contribuíram para uma melhor segurança, usabilidade e manejo da *stack*. Sendo assim, hoje, é preciso que os profissionais de segurança conheçam completamente os sistemas sob ataque, de modo a possuir entendimento pleno do funcionamento da *stack*, bem como das estratégias desenvolvidas para a proteção da mesma.

ESCOPO

A maioria dos conceitos e códigos a serem apresentados, irão considerar a arquitetura x86. Entretanto, ao final do trabalho, serão apresentadas as principais diferenças dessa arquitetura em comparação com a 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.

Para entendermos o processo de criação e manejo da *stack*, devemos dar um passo atrás e entender um pouco sobre o uso básico das *threads* no Windows. Uma *thread* possui alguns componentes que a definem, estando entre eles, a *user-stack* e a *kernel-stack*.

As *stacks* da *thread* fazem parte de um conjunto, o *thread's Context*. É importante saber que no *thread's Context* existem outros elementos. Além disso, esse conjunto de informações, trazidas pelo contexto da *thread*, são diferentes para cada arquitetura, a fim de manter compatibilidade com os diversos sistemas. Vejamos a seguir a função responsável por criar o *thread's Context* de um processo:

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

O parâmetro lpContext é um ponteiro para a estrutura chamada CONTEXT, que contém as informações necessárias para a *thread*. A seguir, a API *CreateThread*, função utilizada para criação de *threads*:

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

O parâmetro **dwStackSize** configura o tamanho da *stack*. Se for passado um valor nulo (zero) no parâmetro **dwStackSize**, a *stack* terá seu tamanho padrão, que é de 1MB. Também é possível alterar o tamanho da *stack* através da *flag* de compilação **/STACK:reserve** presente no Microsoft C/C++. Quando um novo processo é criado, o Windows sempre estabelece, por padrão, o **dwStackSize** como nulo.

Abaixo, uma outra opção utilizada na criação de *threads*, onde o parâmetro **dwStackSize** também configura o tamanho da *stack*:

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

Uma *thread* possui dois tipos de estruturas principais, a nível de atuação do sistema operacional: ETHREAD e KTHREAD. Sendo que a estrutura KTHREAD está contida como primeiro membro da ETHREAD. Através de um *debugger*, pode-se obter as informações da ETHREAD a seguir[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
```

Abaixo, podemos observar a KTHREAD, com os elementos destinados à *stack* em negrito:

```
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
```

[1] Para os exemplos seguintes, foi utilizado o *debugger* Windbg no modo Kernel na máquina local.

```
   +0x058 KernelStack       : Ptr64 Void
   +0x060 StateSaveArea     : Ptr64 _XSAVE_FORMAT
   +0x068 SchedulingGroup   : Ptr64 _KSCHEDULING_GROUP
   +0x070 WaitRegister      : _KWAIT_STATUS_REGISTER
   +0x071 Running           : UChar
   +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
```

Para o aprendizado da *User-Stack*, não é necessário entendermos em profundidade os elementos da estrutura ETHREAD. O importante aqui, é visualizarmos o vínculo que as *stacks* possuem com a *thread*.

Outra estrutura que também carrega informações sobre a *stack* é a TEB (*Thread Environment Block*). Porém, ao contrário das anteriores, esta precisa existir no endereçamento de memória do processo.

No caso abaixo, através do comando **!teb**, é possível visualizar a TEB de um processo com os elementos destinados à *stack* também em negrito:

```
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
```

Realizando o *attach* em um processo e executando o comando "**~**", é possível visualizar todas as *threads* do processo:

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

É possível notar a existência de duas *threads*, a 0 e a 1. Com o comando **~nk**, é possível visualizar a *stack* de cada *thread*, onde "**n**" será o identificador da *thread*.

Veja que o comando **~1k** mostra a *stack* da *thread* número 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
```

As *stacks* são cruciais para o funcionamento das *threads*, e sem ambas é impossível um programa se manter ativo num sistema operacional.

A PRIMEIRA THREAD DE UM PROCESSO

Sempre que abrimos um programa, um novo processo é criado. Durante a criação de cada novo processo, uma *thread* também é feita. Sempre que uma *thread* é executada, um novo espaço na memória é reservado para que sejam alocadas tanto as variáveis locais, quanto os parâmetros e endereços de retorno de chamadas de funções. Além disso, novos *frames* são criados e inseridos na memória, sempre que

uma *thread* realizar uma nova chamada. Um *frame* é, basicamente, a reunião de todos os dados necessários para executar uma função. É justamente a essa reserva de memória, que chamamos de *stack*.

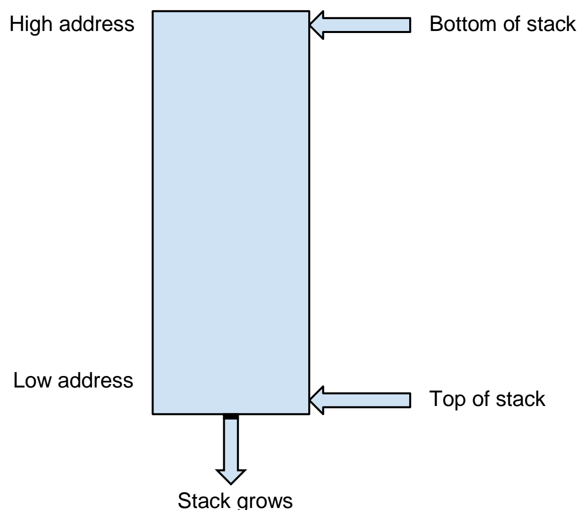
O gerenciador de memória do Windows maneja 3 tipos de *stacks*: a *dpc stack*, a *user stack* e a *kernel stack*. Lembramos que o gerenciador de memória do Windows reservará, automaticamente, 1 MB de memória, para a *User Stack*, no ato da criação da nova thread.

Como o próprio nome já diz, o melhor modo de explicar o funcionamento da *stack* é comparando-a a uma 'pilha' de pratos. Onde os elementos a serem retirados (POP) ou inseridos (PUSH) estarão sempre no topo da pilha; por isso a *stack* é conhecida pela semântica LIFO (*Last In, First Out*).

A figura abaixo ilustra um *layout* sobre a estrutura da *stack* durante uma chamada de função em uma arquitetura x86.

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

A seguir, um desenho mostrando como se desenvolve o crescimento da *stack*. Ilustramos que a mesma começa a partir de algum endereço da memória, crescendo para baixo em direção a endereços inferiores:



Por essa razão, quando falamos em topo da *stack* em x86, falamos sobre o *low address*. Porém, muitos *debuggers* apresentam uma visualização invertida da *stack*.

Para um melhor entendimento do funcionamento da *stack*, podemos carregar um executável para testes, como o **notepad.exe** do Windows, que será nosso exemplo.

Através do comando **x notepad!*main***, é possível visualizar a *main* do programa:

```
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)
```

Para inserir um *breakpoint* na função **notepad!WinMainCRTStartup**, utilizamos o comando **bu**. Já para listar os *breakpoints* inseridos, usamos o comando **bl**, como pode ser observado a seguir:

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

```
notepad!WinMainCRTStartup
```

Assim, ao executar o programa, paramos na primeira função do nosso *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)
```

Uma vez na função **call notepad!__security_init_cookie,** vê-se os valores dos registros:

```
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
```

Em seguida, é possível visualizar os registros ESP e EBP, nas linhas 01 e 02, respectivamente:

```
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
```

A partir da imagem acima, percebe-se que os valores contidos na *stack* tratam-se de instruções escritas pelo processo com a finalidade de criar a primeira *thread* a executar o programa. Além disso, é possível observar o momento em que o EIP (*Extended Instruction Pointer*) está parado sobre a instrução MOV EDI, EDI. Também é possível notar que o registrador EBP (*Extended Base Pointer*) está apontando para a API *BaseThreadInitThunk* da **kernel32.dll**. Por essa razão, esse é um dos primeiros *frames* a ser criado, dada a inicialização da primeira *thread* que acompanha a criação do processo.

Cada vez que uma *thread* realiza uma chamada de função, um novo *frame* é inserido na pilha antes que o programa execute qualquer função; por isso, o sistema operacional executa uma série de chamadas que fazem parte do processo de criação das próprias *threads*.

Sendo assim, é importante ressaltar que, para cada linguagem ou versão do compilador, as funções de iniciação das *threads* podem ser diferentes em seu aspecto, mas não em seu resultado. É importante perceber que, na maioria das vezes, as primeiras funções a serem carregadas em um programa estarão relacionadas à criação de sua primeira *thread*.

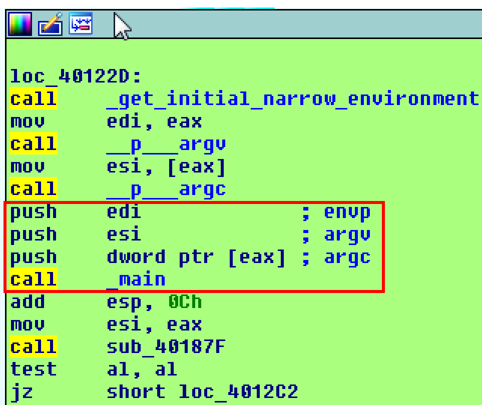
PRÓLOGO, CÓDIGO E EPÍLOGO

Como vimos anteriormente, nem sempre a primeira função de um programa pertence ao seu código. Veja o exemplo de código a seguir:

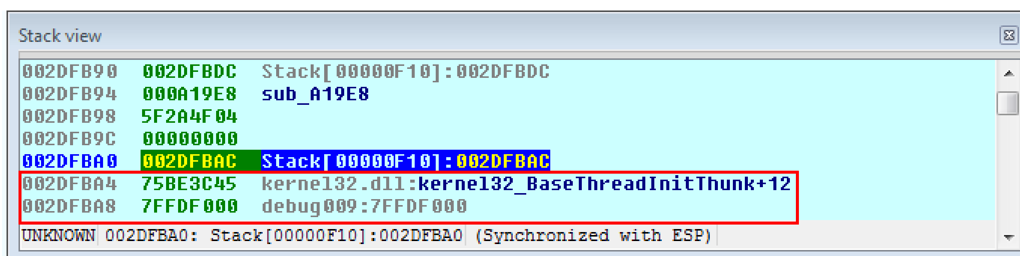
```
int main(int argc, char* argv[])
{
```

```
    char buffer[20];

    printf("Digite qualquer coisa\n");
    gets_s(buffer, sizeof(buffer));

}
```

A imagem abaixo mostra-nos o *Entry Point* do programa e os argumentos que pertencem à *Main* do programa:



Na próxima imagem, é possível visualizar a *stack* no momento em que o EIP está apontado para nossa **call _main**. Observe que abaixo do endereço 0x002DFBA0 está o ponteiro para *BaseThreadInitThunk*, como visto anteriormente:



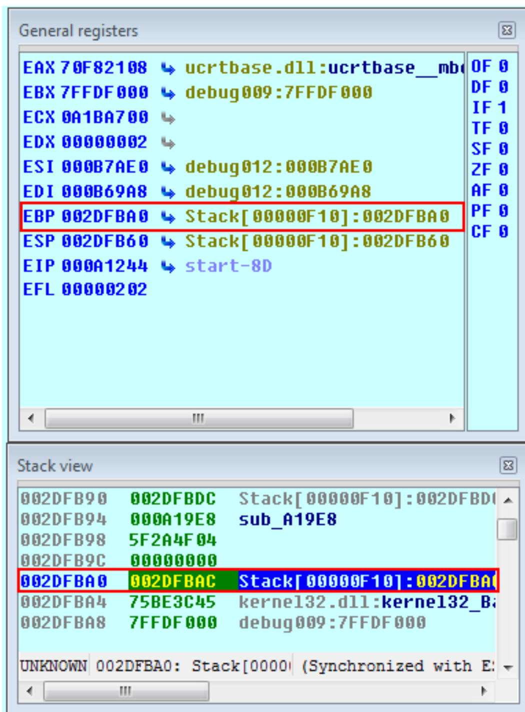
Uma vez dentro da função, é possível visualizar o seguinte conjunto de instruções:

```
; 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
```

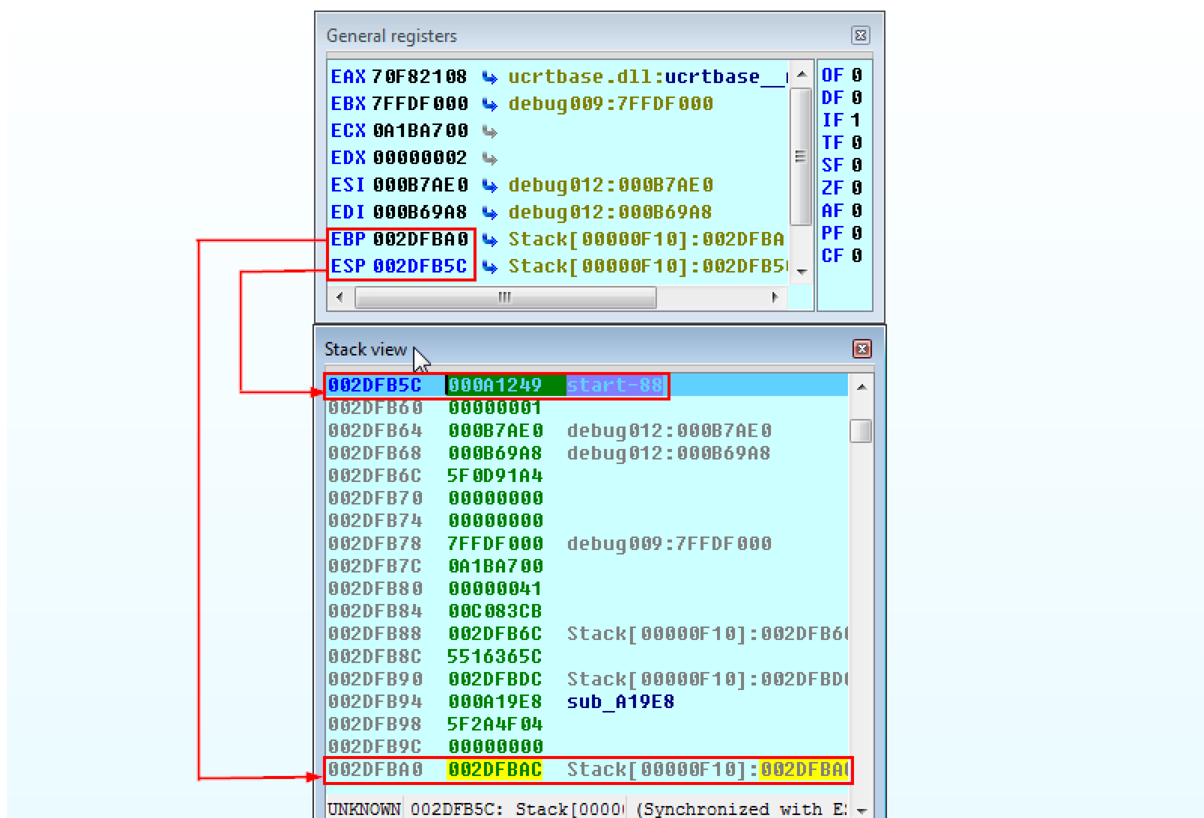
```
.text:000A1040 push    ebp
.text:000A1041 mov     ebp, esp
.text:000A1043 sub     esp, 18h
```

As 3 instruções acima formam o prólogo de uma função. O prólogo de uma função garante que a *stack* esteja preparada para que a nova função seja executada sem problemas.

Antes de explicar o que cada uma das linhas acima significa, precisamos entender o propósito de cada registrador. O registrador EBP aponta para um endereço na memória, geralmente sendo a base do *frame*, com o intuito de facilitar a localização dos argumentos e variáveis de uma função, veja a imagem a seguir:



Já o registrador ESP, é o índice que sempre vai apontar para a parte superior da *stack* (ou o topo da pilha), conforme a imagem a seguir:



Voltando ao conjunto de instruções supracitado, note que a segunda instrução (PUSH EBP) insere o valor de EBP na *stack* através do PUSH. A segunda instrução (MOV EBP, ESP) move o registrador ESP para EBP, pois se o registro ESP for perdido, ainda será possível restaurá-lo através de EBP, estabelecendo assim o início de um novo *stack frame*. Mais adiante, veremos como o registro EBP pode ser utilizado.

A tabela abaixo mostra o estado da *stack* no momento em que passamos pelo prólogo:

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

O endereço de retorno garante a localização do programa para onde se deve retornar ao sairmos da função; pois, quando uma chamada de função é executada, o endereço de retorno da função é inserido (PUSH) na pilha.

Também é importante perceber que a pilha cresce de cima para baixo, decrementado o *stack pointer* (ESP). Note que, na tabela acima, o endereço da *stack* é 0x002DFB5C. Assim, quando o *PUSH EBP* é executado, o valor de ESP é decrementando em 4 bytes (0x002DFB58).

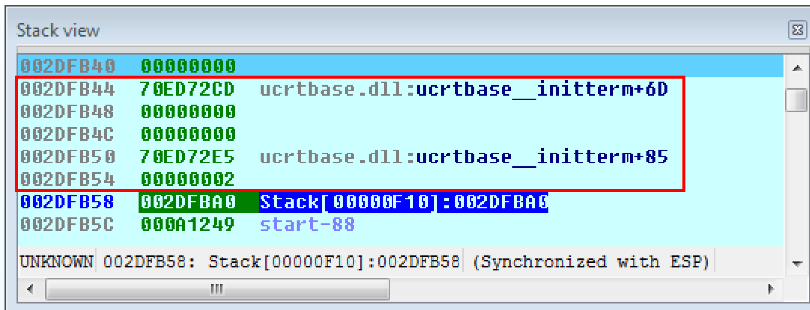
A última instrução (SUB ESP, 0x18) subtrai 18 bytes ou 24 em decimal do *stack pointer*. Essa subtração é importante, pois cria espaço para as variáveis locais.

Observando o *disassembler*, é fácil perceber quais as variáveis locais reservadas:

```
-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)

Quando subtraímos 24 bytes do *stack pointer*, um espaço é reservado na *stack*. A figura abaixo mostra a *stack* depois da subtração:



O novo endereço do *stack pointer* aponta para 0x002DFB40. Sendo assim, ao somarmos 0x18 ao endereço 0x002DFB40, teremos o valor 0x002DFB58. Do mesmo modo que, se subtrairmos 0x18 do endereço 0x002DFB58, chegaremos ao valor do *stack pointer* atual. Olhando a tabela abaixo, é possível observar a *stack* quando a subtração ocorre:

0x002DFB40	Stack Pointer	Sub esp, 0x18
0x002DFB58	Saved EBP	Push EBP
0x002DFB5C	Return address	Call _main

Depois que o *stack pointer* (ESP) foi ajustado para receber as variáveis locais, as próximas instruções relacionadas às variáveis locais foram armazenadas na *stack*. A próxima imagem mostra o relacionamento das variáveis com a *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]
```

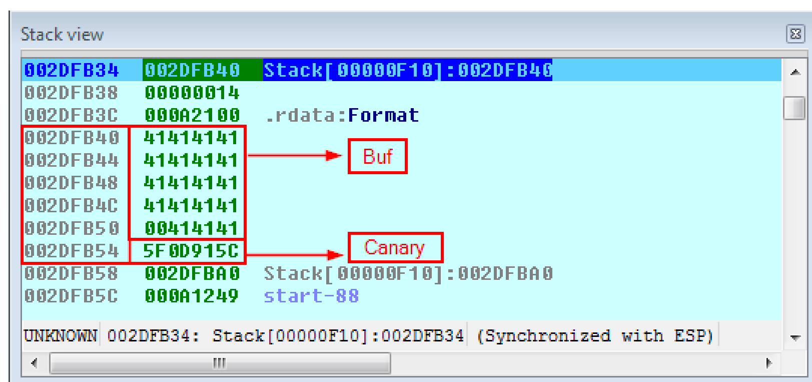
As variáveis CANARY e Buf são armazenadas na *stack* através das instruções:

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

O registro EBP é usado como um ponteiro que faz referência à posição de cada variável local. Essa ação é, por padrão, o propósito do registrador EBP; ao contrário do registrador ESP, que muda continuamente durante a execução da função. Por essa razão, seria difícil usar o ESP como a base do *frame*.

Como vimos anteriormente, o programa possui duas variáveis locais, Buf e Canary. Mais adiante falaremos sobre a variável Canary, pois é uma variável inserida pelo compilador na rotina do programa.

Agora, vejamos como a *stack* se comporta ao receber as duas variáveis com seus respectivos valores:



A tabela abaixo mostra, com mais detalhes, a *stack* no momento em que esta recebe as variáveis:

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

Depois que o corpo do programa é executado, chegamos ao epílogo; cuja responsabilidade é garantir que a *stack* seja restaurada para o ponto anterior à chamada da **CALL _main**. A imagem abaixo mostra o epílogo da função:

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

A primeira instrução soma 0x0C à *stack pointer*. O propósito dessa soma é restaurar a *stack* após os três *pushes* que são os parâmetros utilizados na função, sendo eles: *Buf*, *Canary* e *push offset Format (printf)*. É importante lembrar que, a cada nova função, é preciso restaurar a *stack* ao seu estado anterior.

Para descobrir se o valor acrescido em ESP está correto, basta somarmos os valores em bytes de cada parâmetro em nossa rotina:

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

Após a soma 0xC no *stack pointer*, o novo valor de ESP será o endereço 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

A segunda instrução move o valor de EBP para ESP, lembrando que EBP tem o mesmo valor que ESP possuía, quando armazenada no prólogo. A tabela a seguir, mostra o momento em que ESP vai apontar para a base do *frame*:

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

Tendo em vista que, neste momento, o valor de ESP é igual ao de EBP, a instrução POP EBP deixará o registrador ESP apontando para o endereço de retorno. Além disso, através destas técnicas, o processo consegue reservar o espaço necessário para as funções, bem como para o retorno das mesmas.

HOT PATCHING

Hot Patching é um método de atualização de um sistema ou um programa sem que haja a necessidade de baixar uma nova versão. As atualizações são feitas dinamicamente no executável ou no sistema vulnerável.

Depois de explicarmos as três instruções habituais utilizadas no prólogo, apresentaremos uma quarta instrução, que não raro pode ser encontrada. Essa instrução é o MOV EDI, EDI. Veja abaixo um exemplo de prólogo com ela:

```
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
```

Seria essa instrução parte do prólogo? Como uma instrução que tem efeito nulo, pode ter alguma utilidade prática? A instrução MOV EDI, EDI não possui significado, pois nem mesmo as *flags* são alteradas quando ela é executada. Entretanto, para complicar um pouco mais seu conceito, a Microsoft a define como aquela que "equivale a dois NOP (NO OPERATION), e que possui espaço suficiente para um salto do tipo *short* (**jmp short**)".

Pois bem, a instrução MOV EDI, EDI não pertence ao prólogo da função; e sua utilidade é deixar um espaço reservado para futuras modificações, que seja veloz o suficiente para valer à pena sua existência no programa.

Uma vez que a instrução seja substituída por um *jmp short* (2 bytes), podemos ser levados a um espaço que antecede a primeira instrução do *entry point* daquela rotina, cuja região possui 5 bytes reservados para escrita. Entretanto, diferentemente do salto *short*, o 'salto completo' de 5 bytes é capaz de redirecionar o fluxo do programa para qualquer nova instrução escrita para um *hot patching*.

A imagem adiante mostra os 5 bytes que antecedem nosso *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
```

Caso não houvesse a instrução MOV EDI, EDI, o *hot patching* teria que ser realizado sobre uma instrução que tenha impacto no programa, ou de difícil restauração. O que acarretaria em riscos de correção. Desse modo, sobrescrever uma instrução importante ou difícil de reconstruir não seria uma boa opção. Por esse motivo, a Microsoft decidiu inserir uma instrução a mais, cujo custo de modificação seria nulo, assim como os riscos de alterar partes importantes do programa. Além disso, considerando o tempo de execução, a instrução MOV EDI, EDI é ainda 50% mais veloz que os dois NOPS consecutivos.

Vale lembrar, que se pode adicionar, à aplicação, a opção de *patch* sob demanda, pelo Visual C++, usando a *flag* de compilação **/hotpatch**.

CALLING CONVENTIONS

Quando um código é compilado em Visual C/C++ *calling conventions* são estabelecidas. Em muitos casos, os desenvolvedores menosprezam o assunto, de modo que, na maioria das vezes, discutir sobre o mesmo é desnecessário.

Tradicionalmente, funções feitas em C/C++ possuem o mesmo comportamento em relação à *stack*. Identificar os diversos tipos de *calling conventions* ensina-nos formas distintas de como os argumentos podem ser passados e limpos da *stack*.

Todos nossos exemplos, até então, estão sendo compilados usando a convenção padrão **__cdecl**, onde os argumentos são passados da direita para esquerda, exemplo:

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

Abaixo, um exemplo de chamada:

```
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 soma = num + num2 + num3;

    return soma;
}

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

A tabela abaixo mostra o layout da *stack* durante o uso da função acima:

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	soma [ESP]

Uma segunda convenção, a **__stdcall**, pode ser chamada através da *flag* de compilador **/Gz** que especifica sua utilização para todas as funções. As duas principais características da **__stdcall** são: os argumentos executam-se da direita para a esquerda, e a limpeza da *stack* é feita pela função que a chamou. Sendo assim, é possível criar menores executáveis que a convenção **__cdecl**; pois, com **__cdecl**, a limpeza da *stack* deve ser gerada para cada chamada de função.

Abaixo, um exemplo de código com a convenção **__stdcall** alterada:

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

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


int main()
{
int c = soma(2, 3);

printf("%d", c);

}
```

A terceira convenção, **__fastcall**, é feita de tal modo que alguns argumentos são escritos em registros. Uma das vantagens da **__fastcall** é que as operações realizadas através dos registradores são mais velozes do que pela a *stack*.

Pode-se utilizar a flag de compilação **/Gr** para especificar **__fastcall** para todas as funções. A principal característica da **__fastcall** é que seus primeiros dois argumentos da função são inseridos nos registros ECX e EDX. O restante dos argumentos são inseridos na *stack* da direita para a esquerda.

A tabela seguinte informa as principais diferenças entre os três tipos de convenção:

__cdecl	É a convenção padrão, e sua maior vantagem deve-se ao fato de permitir a existência de um número variável de argumentos. Geralmente, os executáveis tendem a ser maiores.
__stdcall	Não permite um número de argumentos variáveis, sendo possível criar executáveis menores.
__fastcall	Escreve alguns de seus argumentos nos registros, enquanto que os demais são inseridos na pilha. Sua maior vantagem é a velocidade na chamada de funções.

É importante não esquecer, que quando se mistura os tipos de *calling conventions*, pode-se incorrer em alguns problemas, sendo estes chamados de *calling conventions mismatch*[2].

FRAME POINTER OMISSION

O *Frame Pointer Omission* (FPO) é uma técnica que utiliza o *base frame register* (EBP) como um registrador de propósito múltiplo. Geralmente, seu uso está atrelado ao ganho de velocidade; assim, o compilador utiliza o EBP para armazenar vários tipos de dados. O FPO pode causar problemas especialmente para quem está programando diretamente em *assembly*; por isso, deve-se ter cuidado ao usar o *stack pointer*.

[2] Para saber mais sobre *calling conventions mismatch* recomendamos a leitura do blog *The Old New Thing* [https://devblogs.microsoft.com/oldnewthing/20040115-00/?p=41043]

Fazer o *debug* de um programa com FPO pode gerar confusão; pois, caso ocorra um *crash* sem o *frame pointer*, o *debugger* não será capaz de gerar o *stack trace*, a geração apenas ocorrerá caso os símbolos estejam presentes. Por isso, o uso do FPO dificulta o *debug*.

Caso ocorra um *crash* de um executável fazendo uso de FPO, o seu *dump* não irá conter o *frame pointer* da stack. Logo, o *debugger* não será capaz de gerar o *stack trace* corretamente a partir desse *dump*. O *stack trace* pode apenas ser restaurado por completo se os símbolos estiverem presentes no programa. Pois as informações do FPO ficam gravadas no arquivo de símbolos do programa. De todo modo, existe a possibilidade de restaurar esse *stack trace* manualmente; porém, essa tarefa nunca será trivial, além disso, ainda assim, não se pode dizer que ela seja sempre possível.

As imagens seguintes mostram a diferença entre uma rotina de um executável sem FPO e outro com FPO, respectivamente:

```
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
```

Perceba que, na primeira imagem, o uso do registrador EBP é constante. Já na segunda imagem, o registro EBP não aparece na rotina em nenhum momento. O único registrador utilizado como guia na *stack* é o próprio ESP, quando se utiliza o FPO.

Na imagem seguinte, é possível visualizar o EBP quando a aplicação com FPO, encontra-se parada no seu *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
```

A próxima imagem mostra a primeira variável local na *stack*; e, logo abaixo, o endereço de retorno da função. Perceba que, acima do endereço de retorno, não se encontra o *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 // retorno
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
```

Perceba que o FPO não utiliza o registro EBP do modo como vimos anteriormente. Deve-se tomar cuidado ao criar executáveis com o FPO, pois a dependência de um único registrador deixa a *stack* mais suscetível a falhas[3], de tal modo que qualquer corrupção em seus ponteiros causará falha na busca das variáveis locais ou no ponto de retorno. É recomendado ter cuidado quanto à quantidade de lixo que o programa pode escrever na *stack*.

STACK OVERFLOW

Um *stack overflow* ocorre quando uma variável armazenada na *stack* é preenchida por um *buffer* que lhe supere em tamanho, causando uma escrita arbitrária na *stack*. Geralmente, os cenários básicos de explorações que envolvem *stack overflow* são aqueles cujos endereços de retorno são alterados para um endereço de memória controlado.

As variantes que causam um *stack overflow* são diversas, podendo em muitos casos serem frutos de outras vulnerabilidades. Por essa razão, não analisaremos os aspectos técnicos de outras classes de *bug*, concentrando-nos apenas naquelas que fazem referência direta à *stack*. Independente da classe de *bug*, o que nos importa é o comportamento que a *stack* vai apresentar a partir da exploração de uma vulnerabilidade.

[3] Para outro exemplo de falha na *stack* envolvendo FPO recomendamos a seguinte leitura [https://devblogs.microsoft.com/oldnewthing/20040116-00/?p=41023]

Abaixo, um exemplo de *stack overflow*[4]:

```c
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");
}
```

Observando o código acima, vemos o uso da função *gets*[5], cujo defeito é não possuir um valor que controle o tamanho de seu *input*.

A imagem abaixo mostra a rotina principal, sem símbolos:

```
; 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
```

PRÓLOGO

PRINTF

GETS

COOKIE

[4] O exemplo em questão foi retirado de [http://ricardonarvaja.info/WEB/EXPLOITING%20Y%20REVERSING%20USANDO%20HERRAMIENTAS%20FREE/EJERCICIOS/]

[5] Mais sobre a função *gets* pode ser encontrado em [(https://docs.microsoft.com/pt-br/cpp/c-runtime-library/gets-getws?view=msvc-160)]

Na imagem anterior, vemos duas variáveis locais, **var_4** e **var_54** (é possível renomear as variáveis com nomes mais significativos, o que faremos mais adiante). Vejamos agora a *stack* da função *main* e a disposição das variáveis locais, já renomeadas, mais os parâmetros:

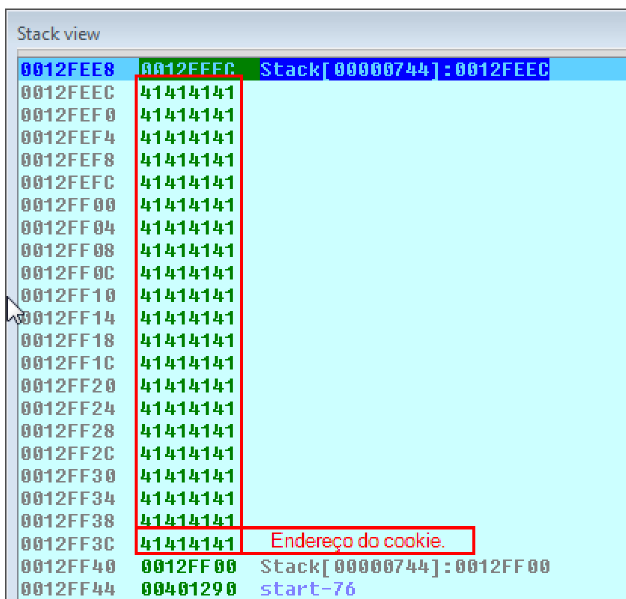
```
Stack of _main
-00000054 ; D/A/*    : change type (data/ascii/array)
-00000054 ; N        : rename
-00000054 ; U        : undefine
-00000054 ; Use data definition commands to create local variables and function arguments.
-00000054 ; Two special fields " r" and " s" represent return address and saved registers.
-00000054 ; Frame size: 54; Saved regs: 4; Purge: 0
-00000054 ;
-00000054
-00000054 Buffer          db 80 dup(?)   Variáveis locais. Buffer = 0x50h.
-00000004 cookie          dd ?
+00000000  s              db 4 dup(?)    S= Stored ebp; R= Return Address.
+00000004  r              db 4 dup(?)
+00000008 argc            dd ?
+0000000C argv            dd ?           Parâmetros    ; offset
+00000010 envp            dd ?                         ; offset
+00000014
+00000014 ; end of stack variables
```

Vamos preencher a *stack* com o valor máximo que a variável *Buffer* espera receber (80 em decimal). A imagem abaixo mostra a *stack* preenchida com 80 caracteres, e a variável *cookie*:

```
Stack view
0012FEE8  0012FEEC  Stack[00000A34]:0012FEEC
0012FEEC  41414141
0012FEF0  41414141
0012FEF4  41414141
0012FEF8  41414141
0012FEFC  41414141
0012FF00  41414141
0012FF04  41414141   Buffer= 80
0012FF08  41414141   caracteres.
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          cookie
0012FF40  0012FF88  Stack[00000A34]:0012FF88          ebp
0012FF44  00401290  start-76                Return
```

Como a função *gets* não limita o tamanho do *input* do usuário, é possível inserirmos mais bytes na *stack* até chegarmos ao endereço da variável *cookie*. Partindo do endereço 0x12FF2C até o 0x12FF3C temos 5 DWORDS, o que resulta em 20 bytes a partir do final esperado do *buffer* até o endereço do *cookie*.

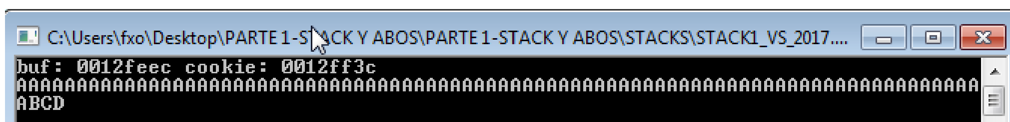
Inserindo 100 bytes, vemos que o valor do *cookie* foi sobrescrito por 41414141:



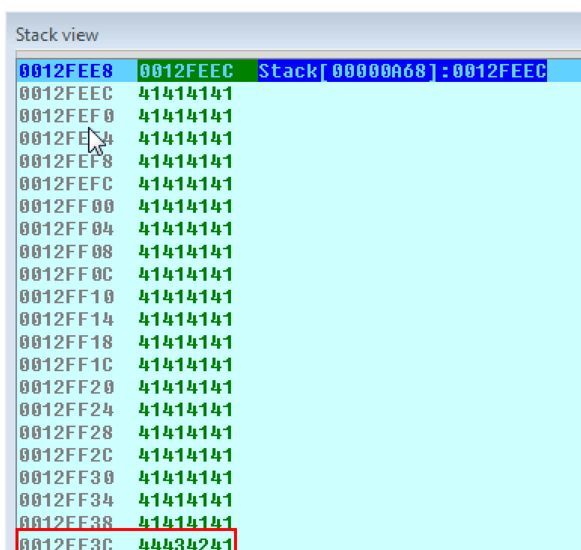
Após observar o código da aplicação, vemos que existe uma condição a ser cumprida:

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

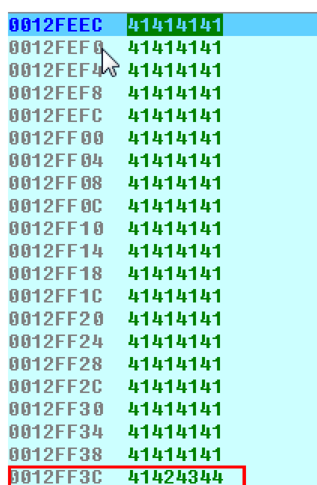
Dessa forma, devemos inserir no fim do *payload* os valores em ASCII de 0X41424344 (ABCD):



E assim, finalmente, observar a *stack*:



Percebe-se que a comparação nunca será correta, pois a ordem dos caracteres é invertida na arquitetura x86. Em outras palavras, a comparação que ocorre nesse momento é: CMP [44434221h], 41424344h. A esse tipo de ordenação chamamos *little endian*[6]. Desse modo, para que nossa comparação aconteça com sucesso, devemos inserir os caracteres finais na ordem invertida, DCBA. Confira na imagem abaixo:



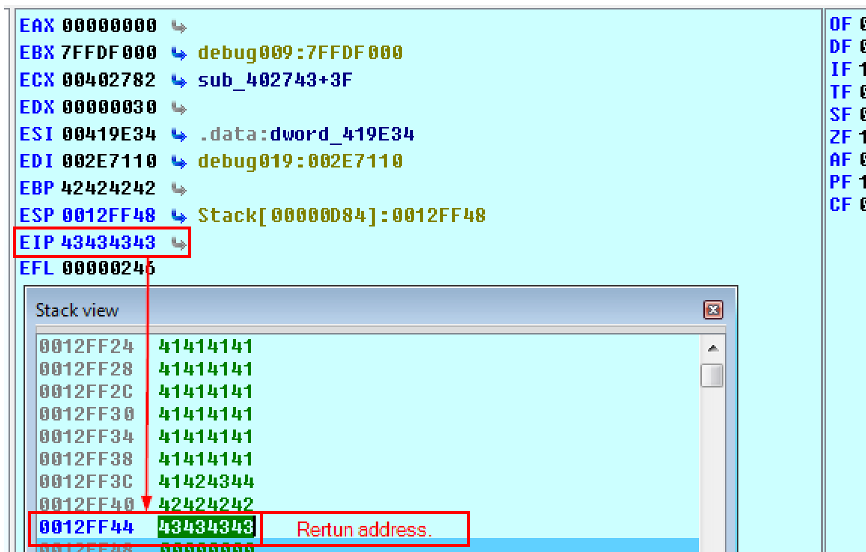
Sendo assim, foi possível redirecionar o fluxo do programa através de um *stack overflow*:

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

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

Da mesma maneira como alteramos o valor do *cookie*, também seria possível continuar com nosso *overflow* até sobrescrever o endereço de retorno.

A imagem seguinte mostrará que, dando continuidade ao *overflow*, é possível alcançar o endereço de retorno. Observe que, nesse momento, o EIP tem o mesmo valor que o endereço de retorno:



Vejamos um segundo exemplo de *stack-overflow*:

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

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

```
}
```

O segundo argumento que o programa irá receber é o **argv[1]**, cujo valor será utilizado como parâmetro da função *copy.* A função *copy*, por sua vez, contém tanto a variável local (*buf*), quanto a função **strcpy**. Esta última, por sua vez, é responsável por copiar o conteúdo do parâmetro *input* para a variável *buf*. A função **strcpy** é uma função insegura, pois não verifica se o *buffer* de destino suportará o valor inserido. Logo, é possível ver claramente que o código da imagem anterior apresenta uma falha de programação.

Vejamos a situação da *stack* depois da execução do programa:

```
0014F710  0014F72C  Stack[00000550]:0014F72C // Endereço do
argumento buf
0014F714  0014F72C  Stack[00000550]:0014F72C // Endereço do
argumento input
0014F718  41414141 // Começo da variável 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 // Fim da variável buf
0014F758  0014F7A0  Stack[00000550]:0014F7A0 // EBP salvo no
prólogo
0014F75C  00851234  start-88 // Endereço de retorno
```

O primeiro ponto a ser controlado pelo nosso *buffer* seria o valor salvo de EBP (0x0014F7A0). Devido à EBP conter o *frame pointer*, caso o atacante controle o valor

de memória 0x0014F7A0, a execução do código seria transferida para a região de memória do ponteiro em questão.

Entretanto, caso o EBP não possa ser controlado, seguimos com a execução sobrescrevendo o valor do endereço de retorno; desse modo, o atacante ainda poderia saltar a execução para qualquer ponto controlado por ele.

Uma vez que o endereço de retorno seja sobrescrito, o atacante também pode começar a explorar o argumento subsequente ao endereço de retorno.

Por fim, concluímos que o conceito de *stack overflow* é bem simples. Por isso, o importante é saber como explorar esse tipo de falha, uma vez que vimos ter sido possível conduzir o fluxo do programa, bem como alterar seu endereço de retorno. Uma vez que aconteça um estouro de pilha (*stack*) é possível explorá-lo de várias maneiras, desde a substituição de variáveis locais, valor de ponteiros de parâmetro; endereços de retorno, e estruturas de exceção[7]; até ataques contra *VTable*, ou mesmo contra um índice de um *array* que não possua seu limite definido.

FLAGS DA STACK

Vimos anteriormente como a *stack* pode ser preenchida facilmente através de um *buffer overrun*, e que nem sempre os resultados de um *overrun* resultarão em *crash*. Com o intuito de manter a integridade da *stack*, foi criado o *canary* (canário), que nos compiladores da Microsoft é conhecido como *guard stack*, cuja *flag* utilizada no compilador é a **/GS**.

A técnica do canário consiste em inserir um valor de checagem entre as variáveis locais e o endereço de retorno. As vantagens são grandes, pois o custo computacional é mínimo, exigindo pouca performance para o executar.

[7] Demonstração de exploração da estrutura de exceção através de um *stack overflow* [https://resources.infosecinstitute.com/topic/seh-exploit/]

A tabela abaixo mostra a *stack* com o uso do canário:

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

Por padrão, o Visual C++ já insere a flag **/GS** no momento da compilação. Vejamos na prática como se comporta o canário em uma aplicação:

```
.data:00403004 ___security_cookie dd 0BB40E64Eh

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

O **__security_cookie** é gerado pelo CRT (*C RunTime*) durante a inicialização, sendo diferente a cada nova execução. Caso a aplicação não utilize o CRT, uma chamada deve ser realizada para **__security_cookie**. Em seguida, será inserido em EAX o valor do **__security_cookie** e um XOR será realizado com o valor de EBP. O valor de EBP também será aleatório a cada nova execução. Além disso, o valor do canário será armazenado em uma posição (DWORD) acima do endereço de retorno.

Utilizando o último exemplo de código, compilado com **/GS**, vemos que o canário está localizado entre a variável local e o *frame pointer*:

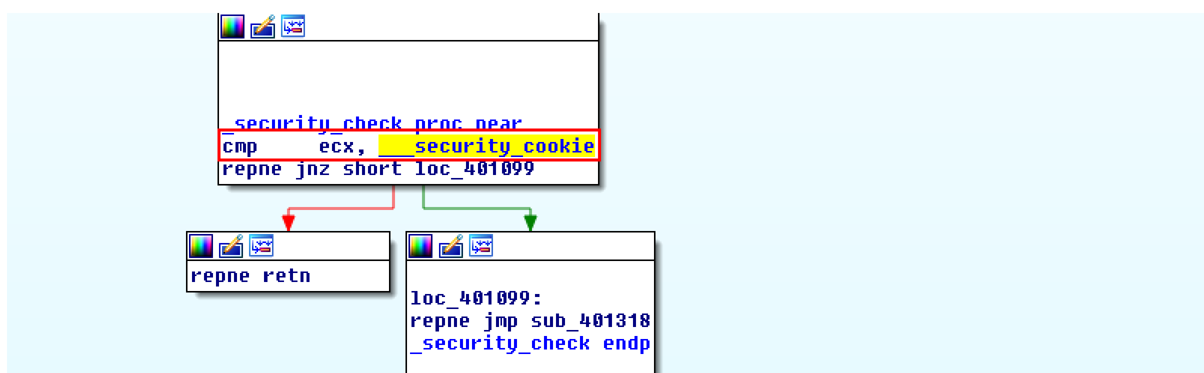
```
-00000044 Buffer          db 64 dup(?)
-00000004 CANARY          dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
```

Na rotina que antecede o endereço de retorno, é possível visualizar o código responsável pela checagem da integridade do valor canário:

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

Ao realizar um XOR entre o valor do canário e o valor original do EBP, é possível restaurar o valor do **__security_cookie**. Nesse caso, sendo esse valor, igual ao criado anteriormente, a variável CANARY não foi sobrescrita, o que garante que o endereço de retorno também não tenha sido alterado. Porém, no caso do valor não ser igual ao original do **_security_cookie**, o programa simplesmente terminará.

A imagem a seguir compara o ECX com o **__security_cookie**:



Com a utilização do canário, estarão protegidos o endereço de retorno, o endereço do *exception handler* de uma função e os parâmetros da função. Porém, o canário não impedirá *buffer overrun* sobre a *stack* e outros tipos de ataque.

Uma outra *flag* importante é a **/NX**, conhecida como DEP[8] (*Data Execution Prevention*). Ela impede que certas regiões de memórias sejam executáveis, desse modo mesmo que um atacante conseguisse escrever dados maliciosos na pilha, ele não os executaria, pois a região de memória da *stack* estaria protegida contra execução de código[9].

Uma terceira *flag*, também importante, é **/RTC** (*RunTimeChecks*). A RTC realiza algumas checagens essenciais. A exemplo:

- **/RTCs**: Checa, em tempo de execução, erros na *stack frame*. Cada vez que uma função é chamada, ele inicializa todas variáveis locais com valores aleatórios, a fim de evitar valores de chamadas anteriores.
- **/RTCc**: Fornece proteção contra perda de dados. Um exemplo dessa perda seria um *casting* do tipo ULONG para um BYTE, onde possivelmente dados serão perdidos. Essa checagem, no compilador, vai mostrar uma mensagem de erro sempre que um *cast* resultar em perda de dados.
- **/RTCu**: Fornece proteção para variáveis não inicializadas. O compilador sempre mostrará um erro sempre quando uma variável for acessada, antes da sua inicialização.

A *flag* de compilação **/RTC** foi projetada para trabalhar com *builds* no modo *debug*. Portanto, as checagens realizadas pelo RTC não funcionam em programas no modo *release*.

SHADOW STACK

[8] A partir do Windows 7 a DEP já se encontra ativada por padrão.
[9] Entretanto, existe um método para burlar o DEP, para mais detalhes ver [https://fluidattacks.com/blog/bypassing-dep/]

A Intel, a partir de 2016, implementou em alguns de seus *chipsets*, o que eles chamaram de CET (*Control-Flow Enforcement Technology*). Uma tecnologia cuja finalidade é proteger os usuários de ataques do tipo *control-flow hijacking*. Porém, apenas o Windows 10 dá suporte para essa tecnologia.

As técnicas implementadas pelo CET são: *Shadow Stack* e *Indirect branch tracking*. Mas como este documento é sobre a *stack*, iremos abordar somente a implementação da *Shadow Stack*. A Microsoft decidiu chamar esta técnica de *hardware-enforced stack protection*; mas alguns pesquisadores de segurança também se referem a ela como *Return Flow Guard* (RFG).

Quando o CET é ativo, um novo registro é usado, o SSP (*Shadow Stack Pointer*). O registro SSP não pode ser utilizado com os mesmo propósitos da *stack* original.

Tal como o registro ESP, que aponta para o topo da *stack*, o registro SSP aponta para o topo da *shadow stack*.

Quando o *shadow stack* está ativo, e o programa encontra-se próximo a executar uma função, nesse momento, são inseridos em ambas *stacks* o endereço de retorno. Se o endereço de retorno não for o mesmo em ambas *stacks*, o processador gerará uma exceção (INT 21 - *Control Protection Fault*).

A técnica *shadow stack* nada mais é que um *backup* do endereço de retorno para cada função. Caso o endereço de retorno seja sobrescrito, no fim da rotina será realizada uma comparação com a *stack shadow* para se comparar os valores de retorno. Caso os valores não sejam os mesmos, o programa será finalizado.

Depois de ter sido descontinuado pela Microsoft, o *Return Flow Guard* (RFG[10]) foi relançado este ano. Entretanto, esse tipo de implementação apenas pode ser vista nos processadores de 11ª geração da intel, chamados de *Tiger Lake*[11].

O QUE MUDA NO x64?

Nesta seção, abordaremos as principais diferenças entre as *stacks* das arquiteturas x86 e x64.

No *debugger*, os valores em 64-bits são representados através de dois números de 32-bits, e algumas vezes separados por um acento grave (`). Por exemplo, 0x60000000`00000000. Esse valor sendo equivalente a 0x60000000 nos processadores x86.

Os registros também foram estendidos na arquitetura x64. Os registradores agora começam com a letra "**r**" ao invés da letra "**e**". Os mnemônicos do x86 ainda estão mantidos. Por exemplo, EBX ainda existe e equivale aos 32 bits menos significantes do registro RBX. Além disso, outros 8 registros foram adicionados, indo do registro **r8** ao registro **r15**.

A seguir podemos ver os registradores em uma arquitetura x64:

```
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] Para mais informações [https://windows-internals.com/cet-on-windows/]
[11] Idem [https://en.wikipedia.org/wiki/Tiger_Lake_(microprocessor)]

No Windows x64, temos apenas um tipo de convenção:

```
rcx: Contém o primeiro parâmetro passado à função.
rdx: Contém o segundo parâmetro passado à função.
r8: Contém o terceiro parâmetro passado à função.
r9: Contém o quarto parâmetro passado à função.
rax: Contém o resultado de uma função.
```

Caso uma função possua mais que 4 parâmetros, os mesmos serão guardados na *stack* da direita para esquerda. Sendo que o parâmetro mais à direita será sempre guardado primeiro na *stack*.

A tabela abaixo mostra o *layout* de uma *stack* em 64-bits:

Parâmetros destinados à *stack*
R9
R8
RDX
RCX
Return

CONCLUSÃO

Ao longo desse artigo, vimos que a *stack* é alinhada e organizada através do prólogo, do código e do epílogo. Além disso, estudamos também o modo como os diferentes tipos de *calling conventions* podem influenciar nosso entendimento da *stack*. Aprendemos como o *frame point omission* é capaz de tratar a *stack* apenas com o registrador ESP. Por fim, estudamos as *flags* de compilação que envolvem a

segurança da *stack,* tal como o canário e a *shadow stack*. Além disso, devido à diferença entre as arquiteturas, apresentamos as mudanças da versão x64.

Concluímos lembrando que os modos como as stack são construídas e mantidas, através de prólogos e epílogos baseados em diferentes arquiteturas ou até mesmo através de *flags* de compilação, podem ter suas peculiaridades ou diferenças; mas no final das contas, o que um atacante sempre vai tentar fazer será sobrescrever parâmetros, variáveis locais ou endereços de retorno contidos na pilha, não importando como as informações foram parar lá.