ROP Attack

BYPASSING MEMORY PROTECTIONS USING RETURN ORIENTED PROGRAMMING

PRESENTED BY AHMAD MOGHIMI

What is ROP Attack?

- Return Oriented Programming attack is a technic to bypass some memory protection.
- Can also be used to bypass antivirus detection mechanism.
- JOP (Jump Oriented Programming) is another variation of this attack technic.
- Formally we call such technics as Code Reuse attacks.
- Memory protection is a kind of protection designed and implemented in OS and 3rd party protection software to block defend against memory corruption attack.

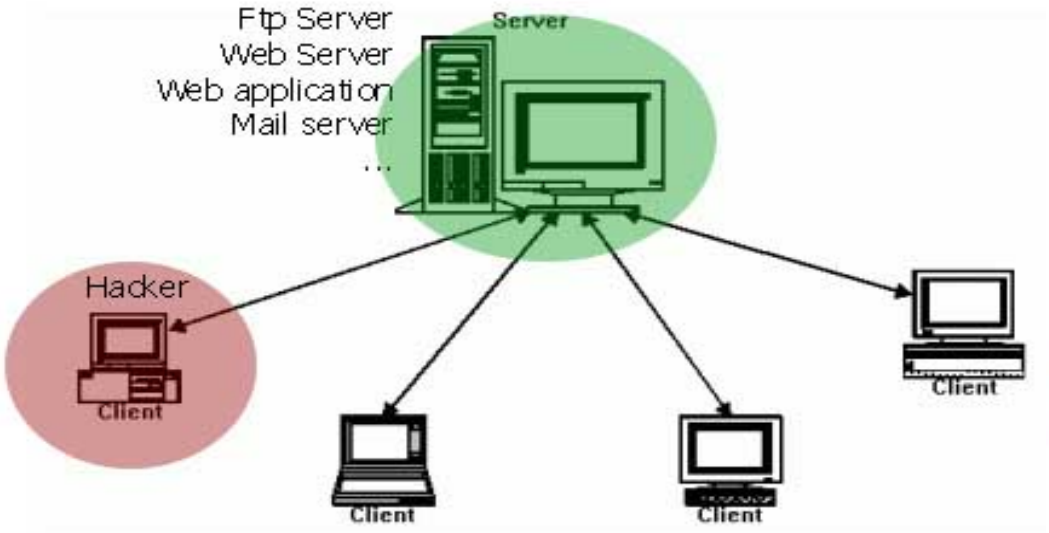
Memory Corruption Attack

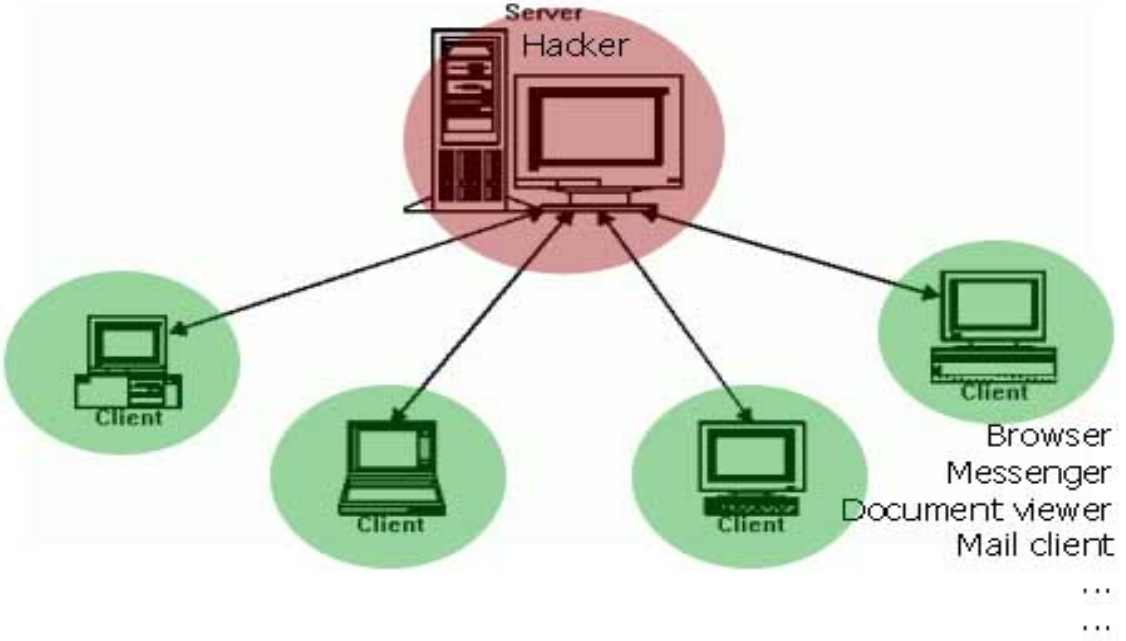
- Programming languages like C, C++ compiles to machine code.
- The programmer is responsible to manage memory via memory management APIs and pointers.
- It is more efficient but less secure.
- Operating Systems, Browsers, Web servers, Games, and almost any critical software that you run everyday is developed using these languages and compiled in to machine code.
- Memory corruption occurs when programmer don’t manage memory appropriately especially in exceptional conditions.
- Memory corruption attack occurs when programmer don’t process user input appropriately in exceptional conditions.

Memory Corruption Attack – Cont'd

- There are different type of memory corruption vulnerabilities based on the condition and type of memory which is targeted.
  - Buffer Overflows
    - Stack
    - Heap
    - …
  - Integer Overflow
  - Dangling Pointer
- The attacker abuse these vulnerabilities to execute arbitrary machine code in the context of the running application by sending malicious user inputs. (Exploitation)

Exploitation



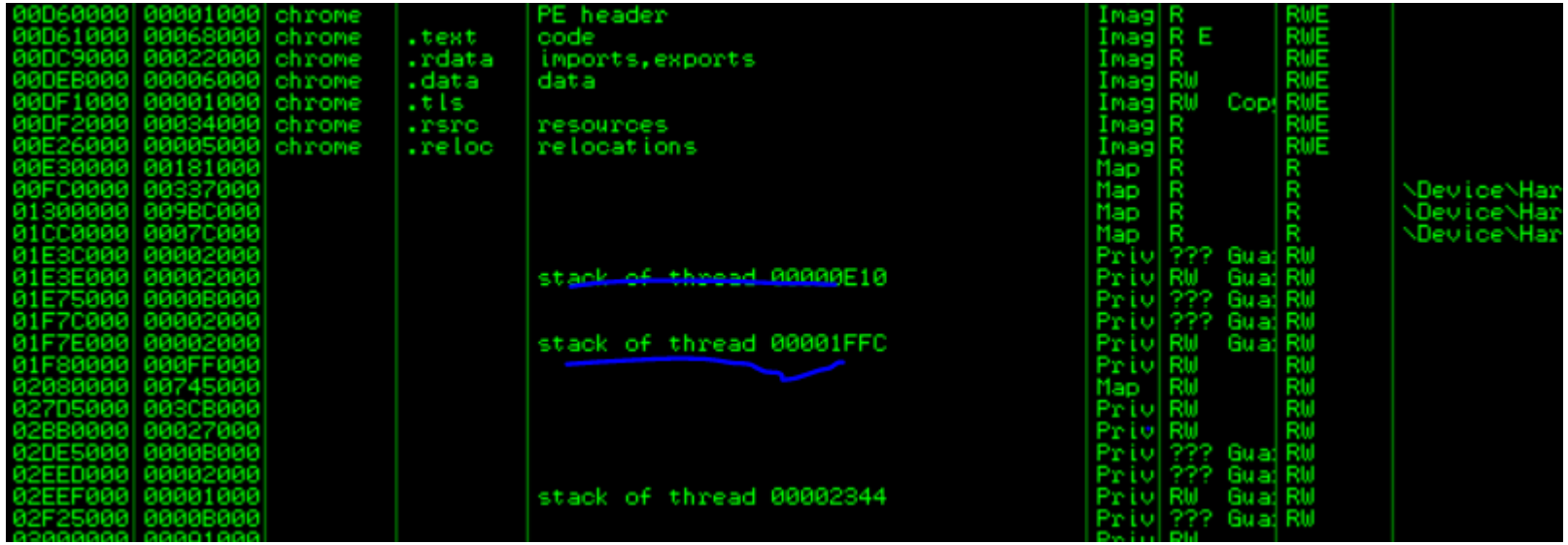


Software Exploitation Terms

- Application: Binary application (No high level application)
  - Compiled to machine code
  - Machine code: Binary code can be reverse engineered to assembly language.
  - Can be debugged by low level debuggers(Immunity debugger, WinDebug)
- Exploitation: Binary application exploitation
  - Abuse user input to cause memory corruption condition in the targeted application.
  - Exploit the condition to execute delivered machine code through user input as shellcode.
- Shellcode: Payload
  - Independent machine code
  - Delivered through user input
  - Do malicious activity like Download & execute malware, Open backdoor ports, …

Stack

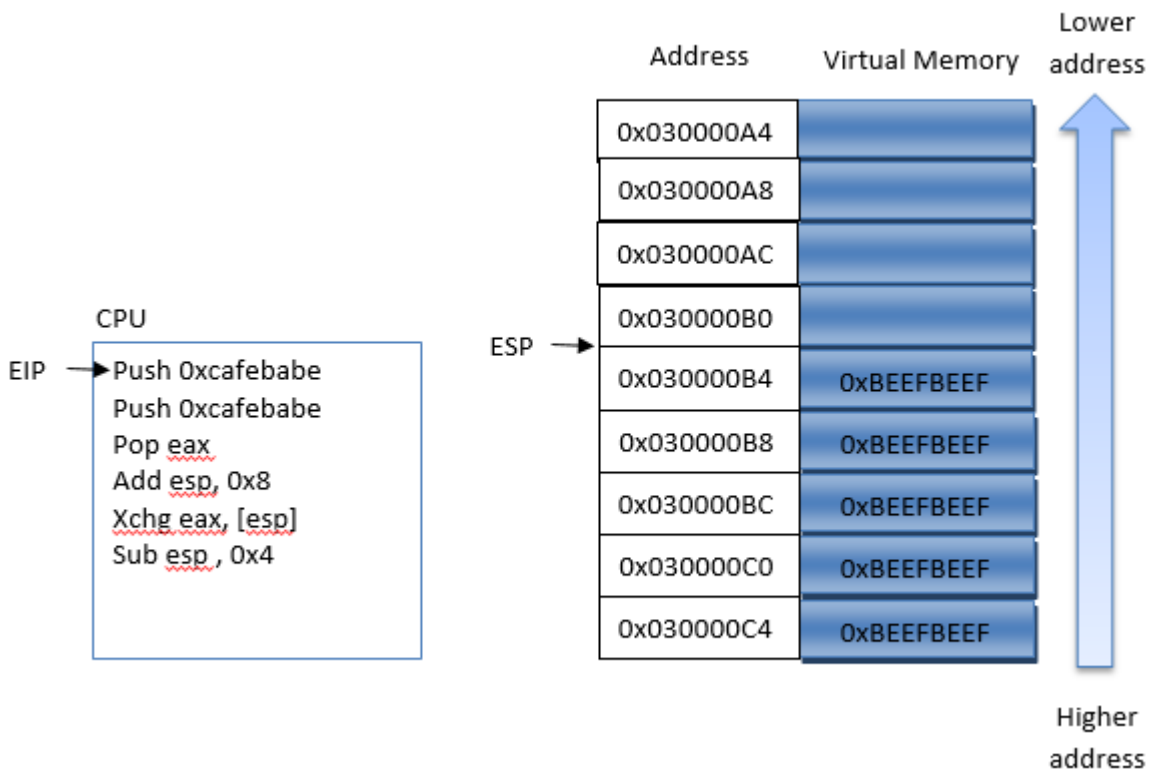
- Stack is a kind of memory used by applications.
- Can store user input data, so malicious user data, so can be exploited!
- From now one, We explain everything in the context of x86 CPU.



Stack – Cont'd

Stack:

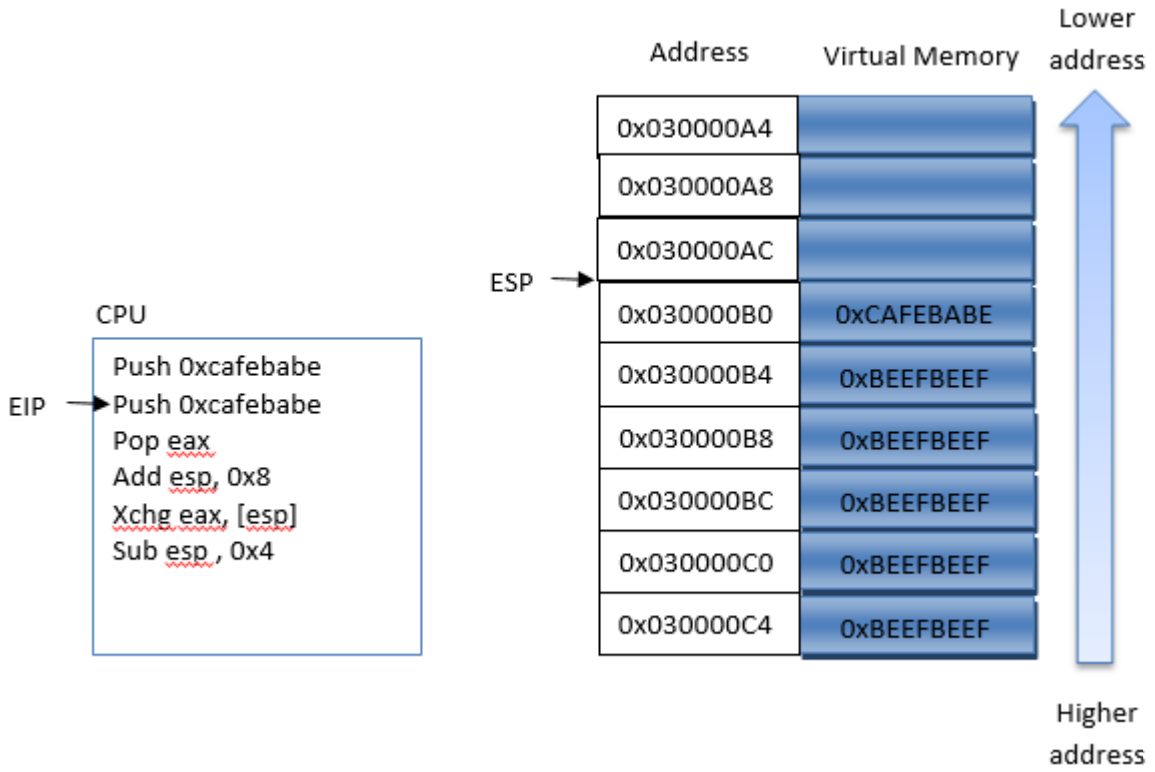
- LIFO (Last-In First-Out)
- Grows to lower address
- PUSH/POP instructions
- ESP Register



Stack – Cont'd

Stack:

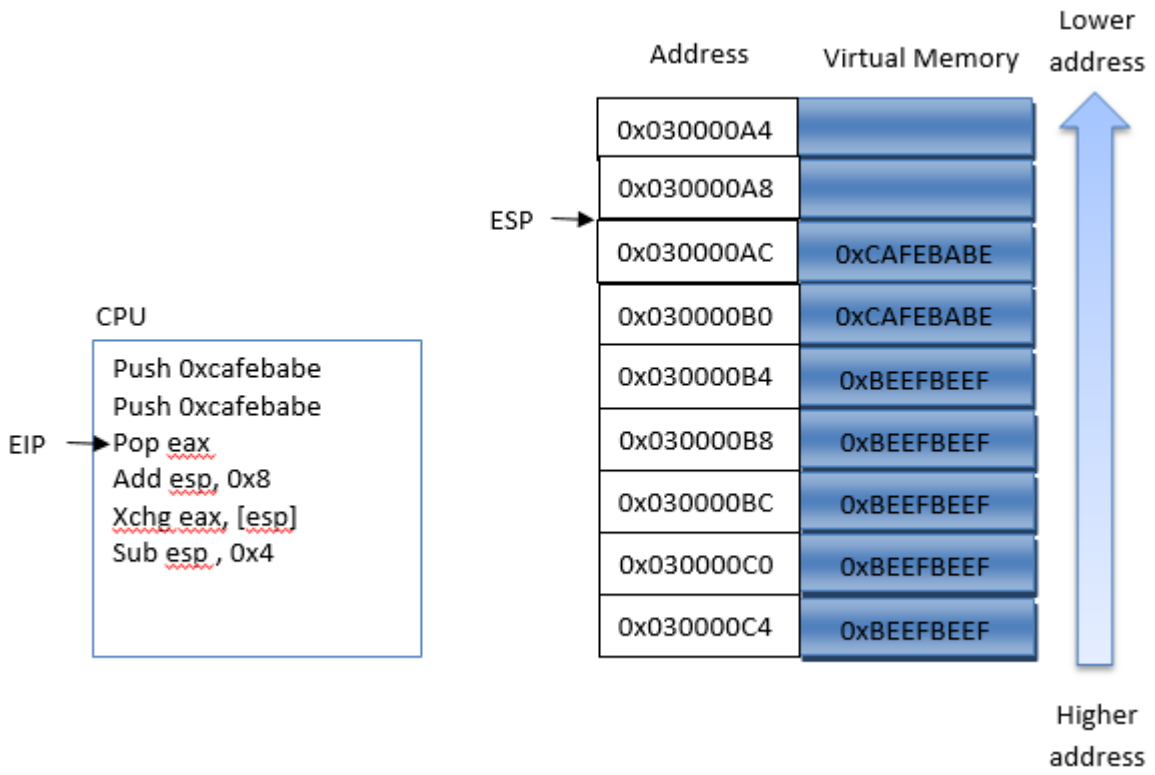
- LIFO (Last-In First-Out)
- Grows to lower address
- PUSH/POP instructions
- ESP Register



Stack – Cont'd

Stack:

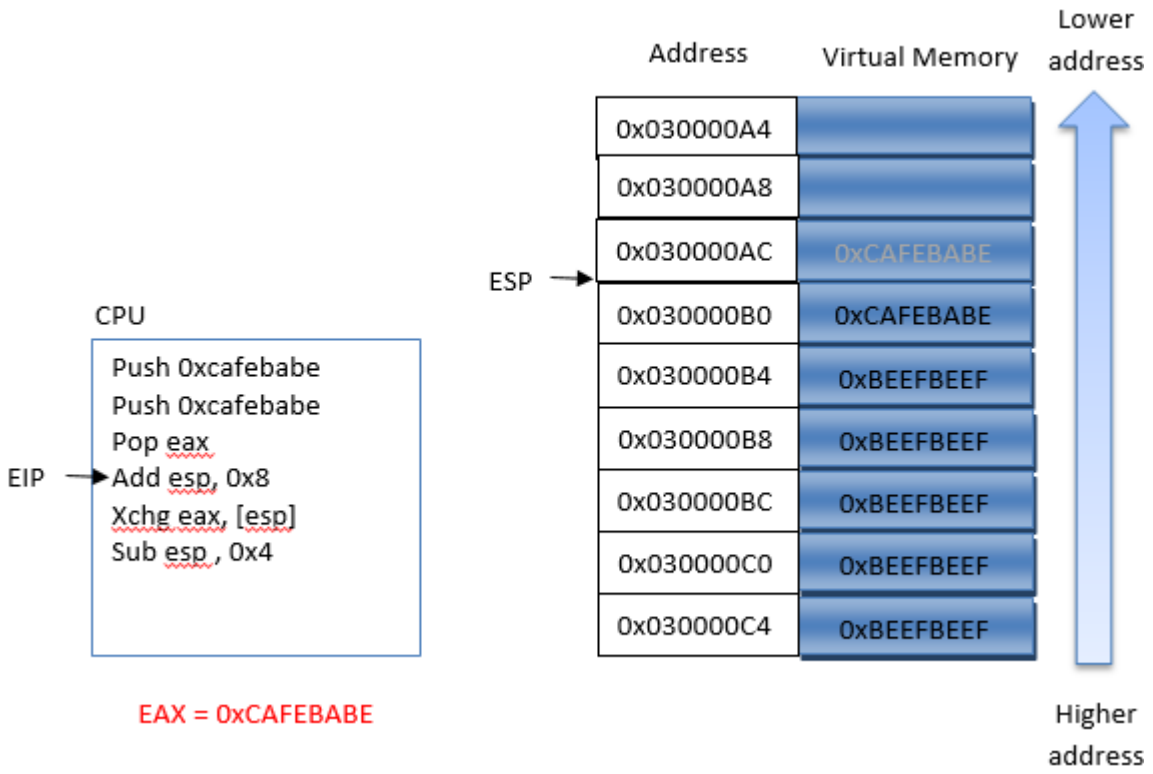
- LIFO (Last-In First-Out)
- Grows to lower address
- PUSH/POP instructions
- ESP Register



Stack – Cont'd

Stack:

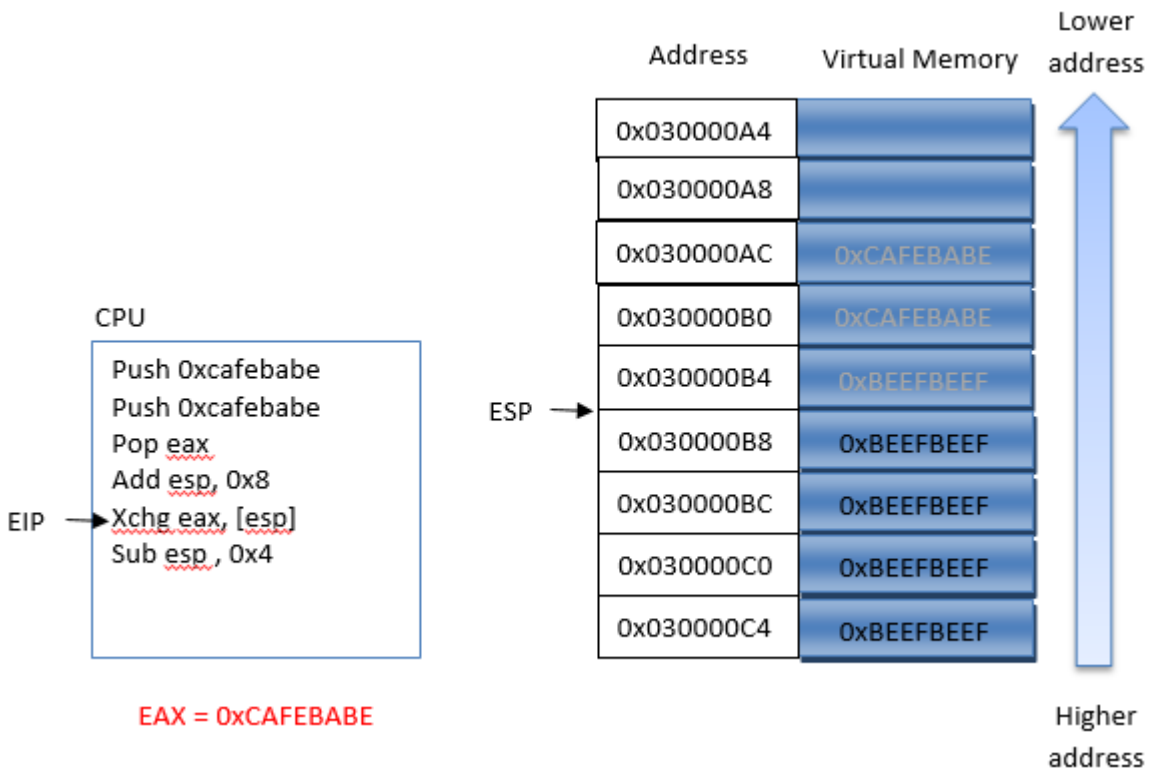
- LIFO (Last-In First-Out)
- Grows to lower address
- PUSH/POP instructions
- ESP Register



Stack – Cont'd

Stack:

- LIFO (Last-In First-Out)
- Grows to lower address
- PUSH/POP instructions
- ESP Register



Stack – Cont’d

- Every thread of execution has two stack
  - User-mode stack
  - Kernel-mode stack
- To store various things
  - Local variable
  - Subroutine parameter
  - Return address in function calls
  - Current state of registers

Stack – Cont’d

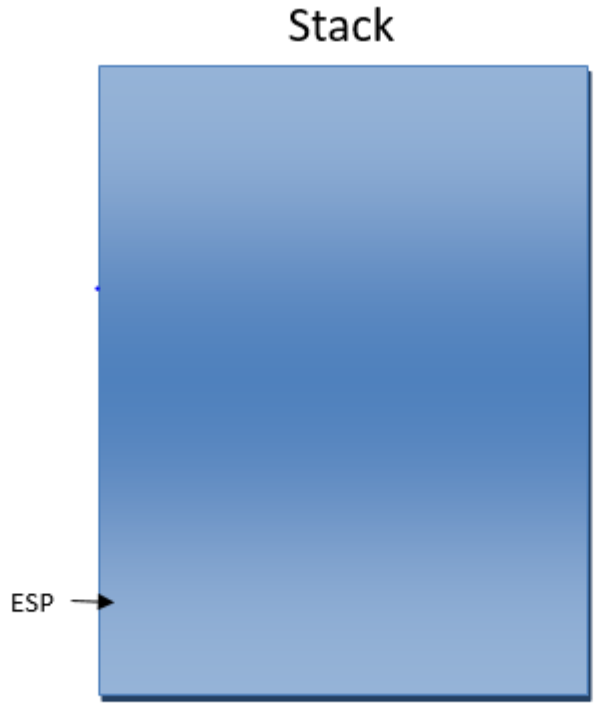
- Function call and stack steps:
  1. Push parameters on the stack based on calling convention
  2. Push Return address on the stack
  3. Jump to the function code
  4. Execute Function prologue
  5. Execute the actual subroutine
  6. Execute Function epilogue
  7. Fetch the return address from the stack to EIP

Stack – Cont'd

Machine code

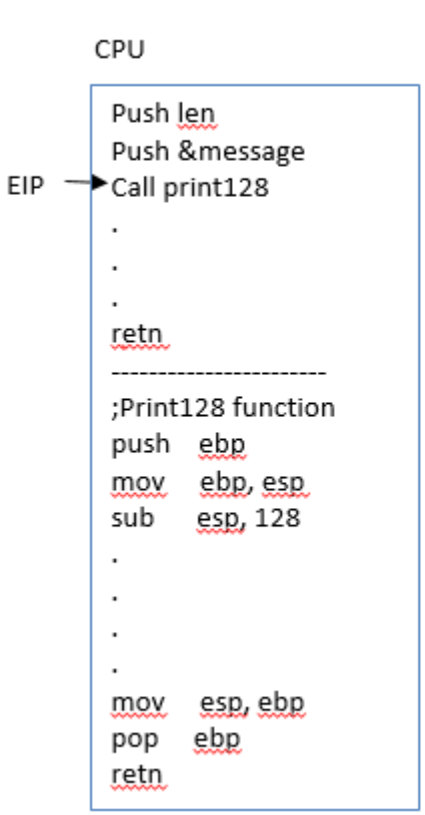
EIP → Push len
Push &message
Call print128
.
.
.
retn

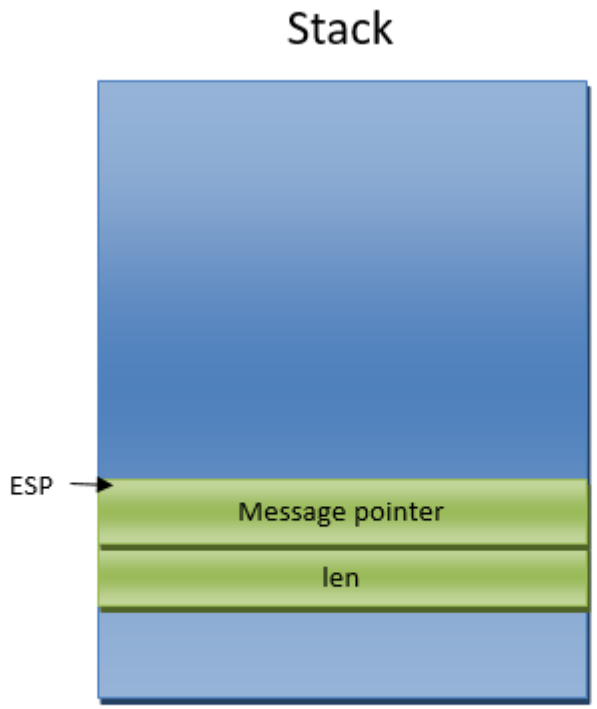
;Print128 function
push ebp
mov ebp, esp
sub esp, 128
.
.
.
.
mov esp, ebp
pop ebp
retn




```
#include <stdio.h>

void print128(char *input, INT32 size)
{
   char buffer[128] = "";
   if( size + 1 > 128)
     return;
   strcpy(buffer, input);
   printf(buffer);
}
int main()
{
   char message[] = "Exploiting!";
   int len = strlen(message);
   print128(message, len);
}
```

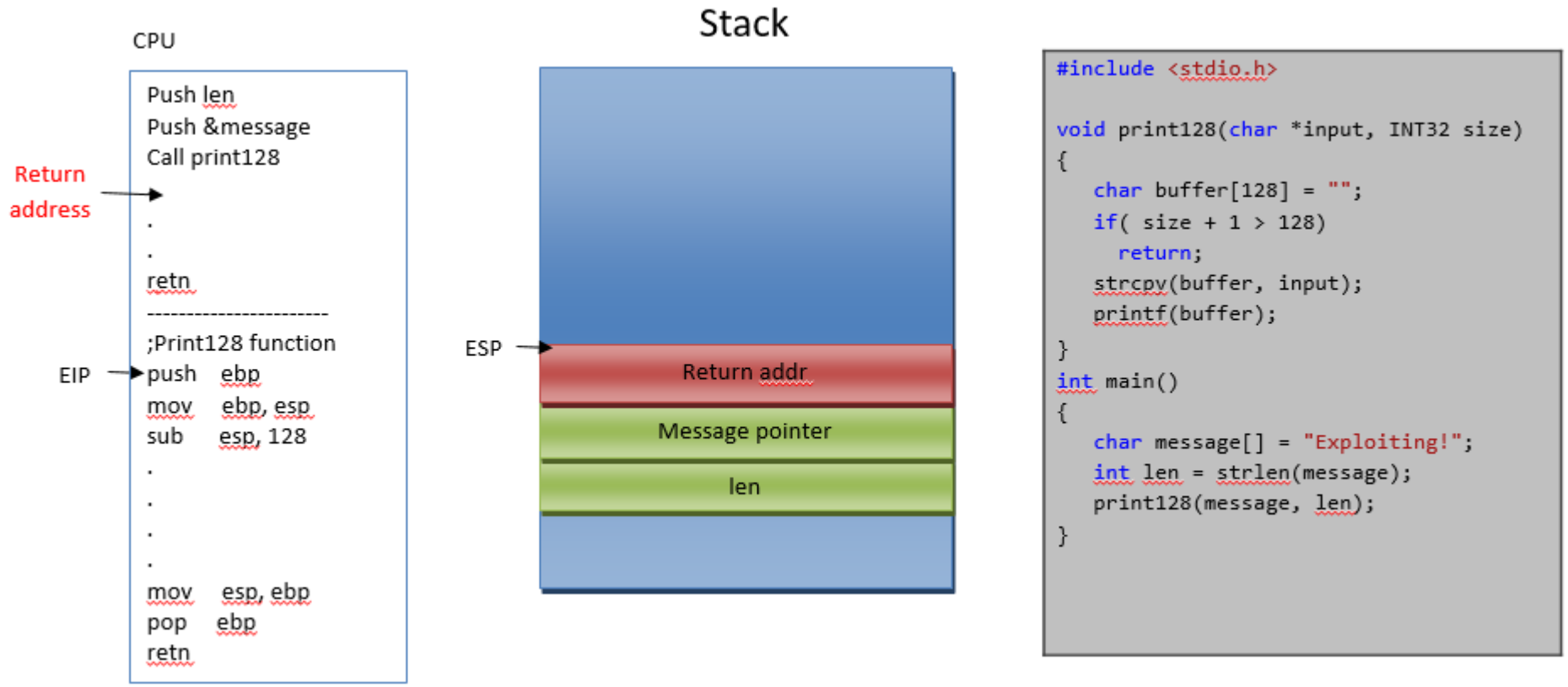
Stack – Cont'd





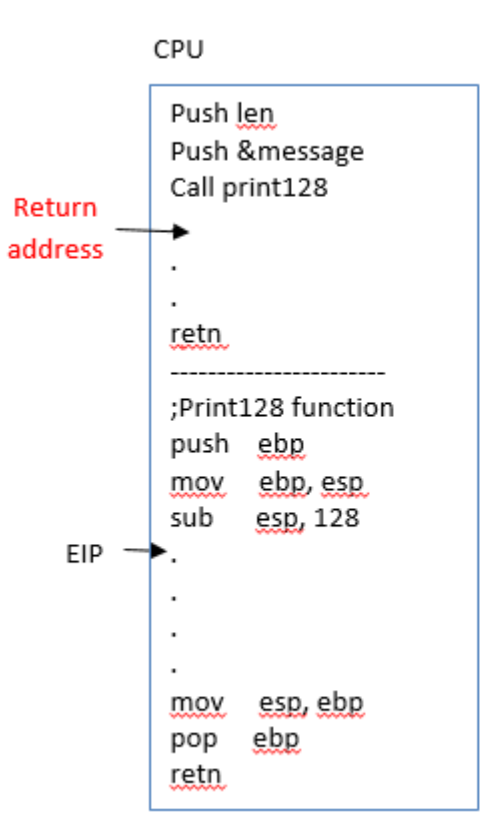

```c
#include <stdio.h>

void print128(char *input, INT32 size)
{
   char buffer[128] = "";
   if( size + 1 > 128)
     return;
   strcpy(buffer, input);
   printf(buffer);
}
int main()
{
   char message[] = "Exploiting!";
   int len = strlen(message);
   print128(message, len);
}
```

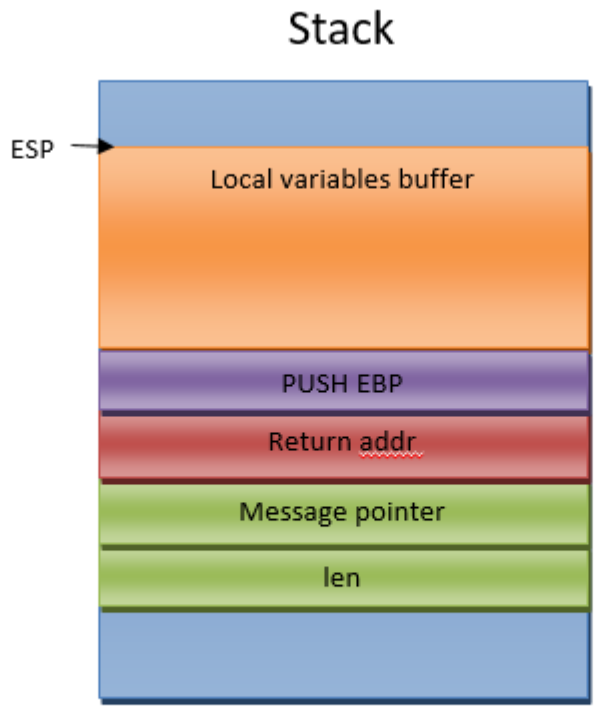
Stack – Cont'd




```
#include <stdio.h>

void print128(char *input, INT32 size)
{
   char buffer[128] = "";
   if( size + 1 > 128)
     return;
   strcpy(buffer, input);
   printf(buffer);
}
int main()
{
   char message[] = "Exploiting!";
   int len = strlen(message);
   print128(message, len);
}
```

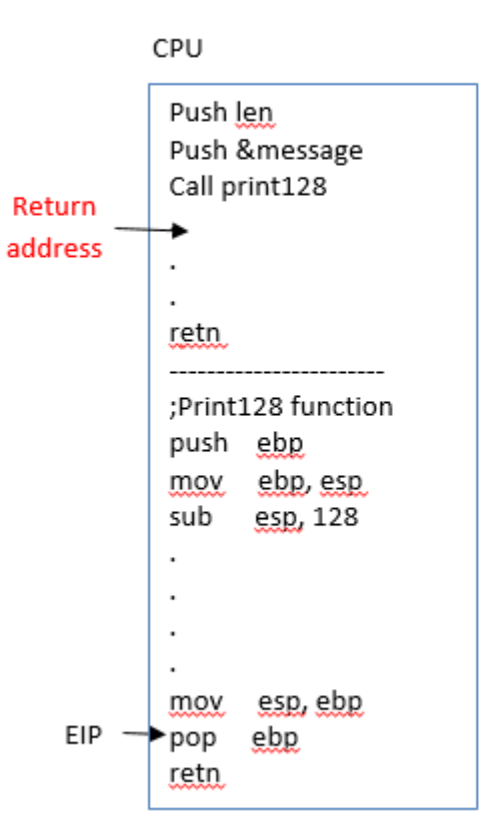
Stack – Cont'd

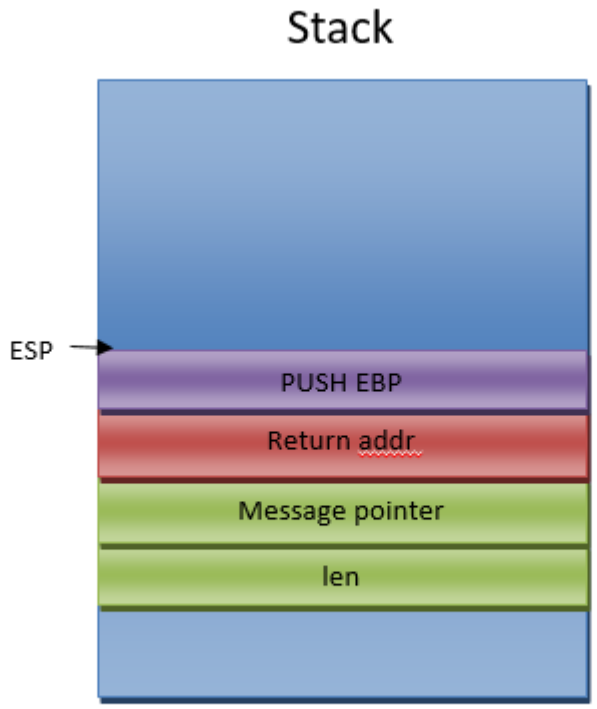





```c
#include <stdio.h>

void print128(char *input, INT32 size)
{
   char buffer[128] = "";
   if( size + 1 > 128)
     return;
   strcpy(buffer, input);
   printf(buffer);
}
int main()
{
   char message[] = "Exploiting!";
   int len = strlen(message);
   print128(message, len);
}
```

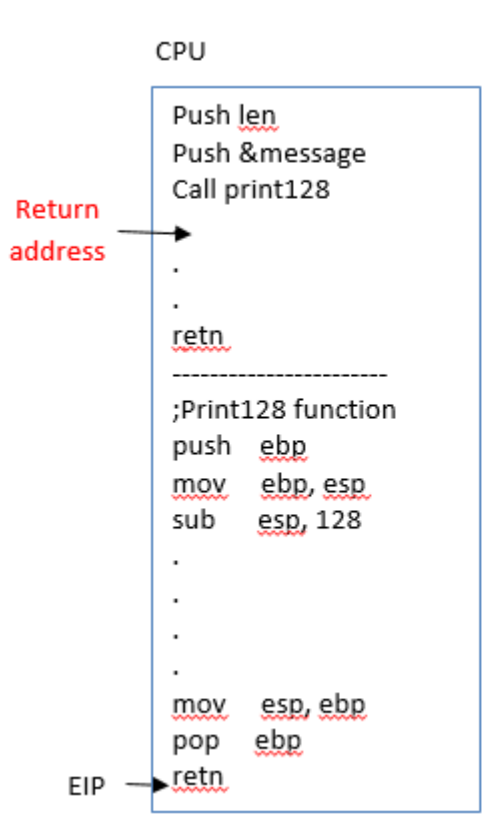
Stack – Cont'd

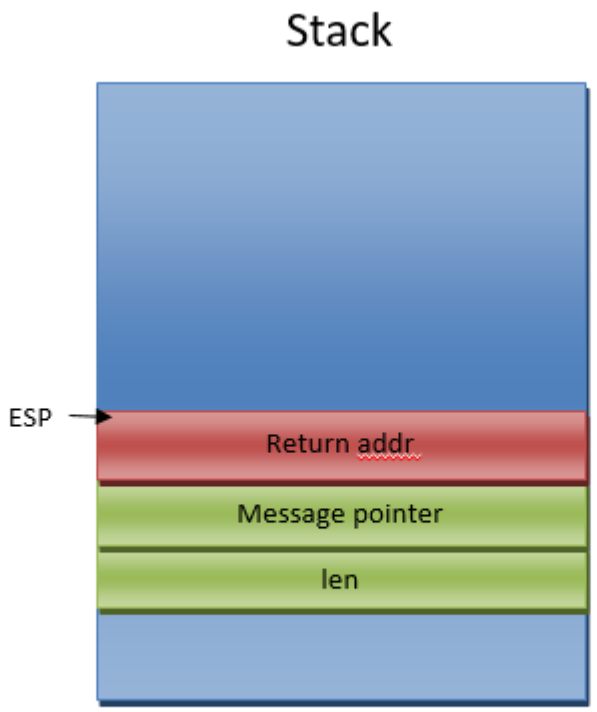





```c
#include <stdio.h>

void print128(char *input, INT32 size)
{
   char buffer[128] = "";
   if( size + 1 > 128)
     return;
   strcpy(buffer, input);
   printf(buffer);
}
int main()
{
   char message[] = "Exploiting!";
   int len = strlen(message);
   print128(message, len);
}
```

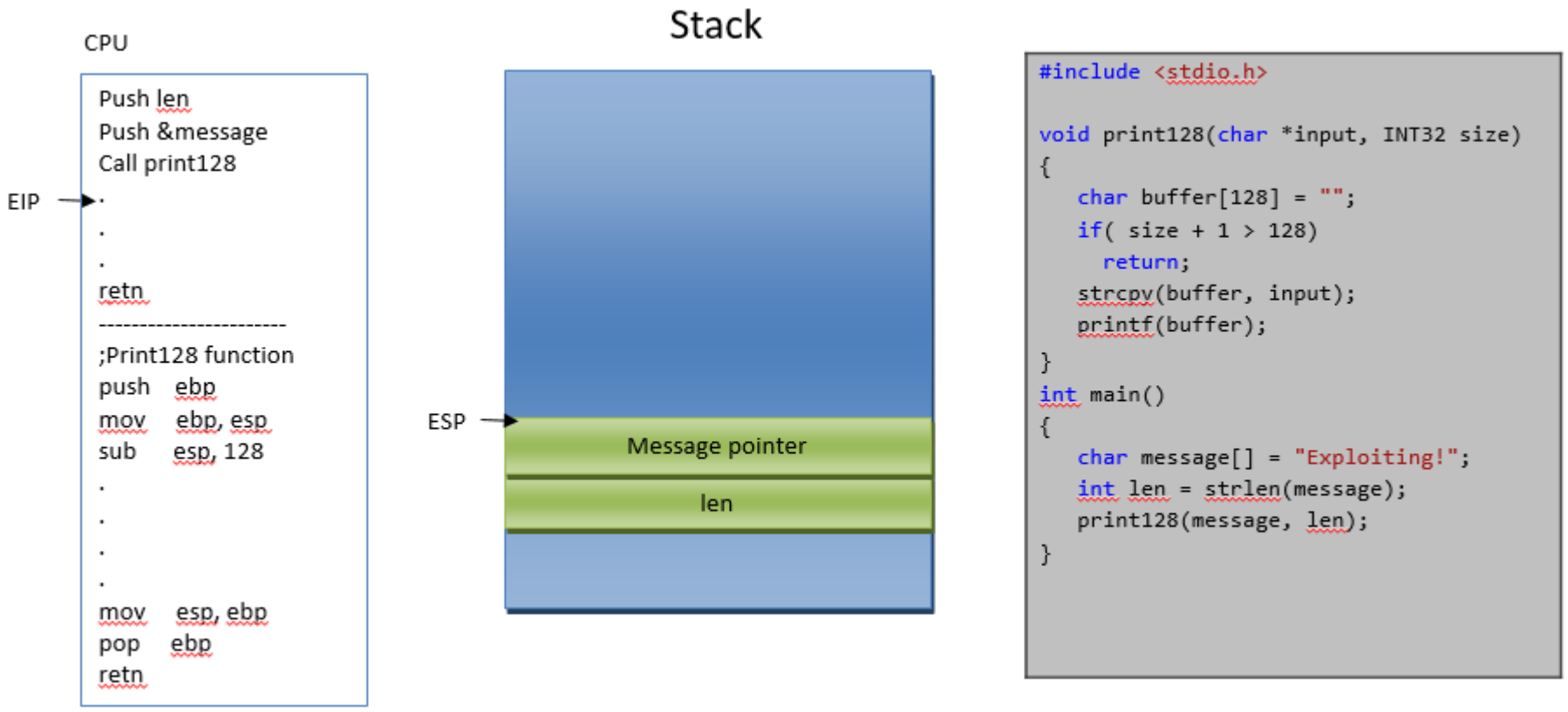
Stack – Cont'd






```c
#include <stdio.h>

void print128(char *input, INT32 size)
{
   char buffer[128] = "";
   if( size + 1 > 128)
     return;
   strcpy(buffer, input);
   printf(buffer);
}
int main()
{
   char message[] = "Exploiting!";
   int len = strlen(message);
   print128(message, len);
}
```

Stack – Cont'd



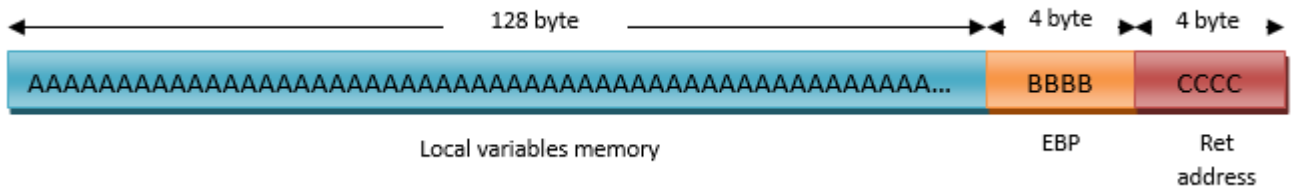

```c
#include <stdio.h>

void print128(char *input, INT32 size)
{
   char buffer[128] = "";
   if( size + 1 > 128)
     return;
   strcpy(buffer, input);
   printf(buffer);
}
int main()
{
   char message[] = "Exploiting!";
   int len = strlen(message);
   print128(message, len);
}
```

Calling Convention

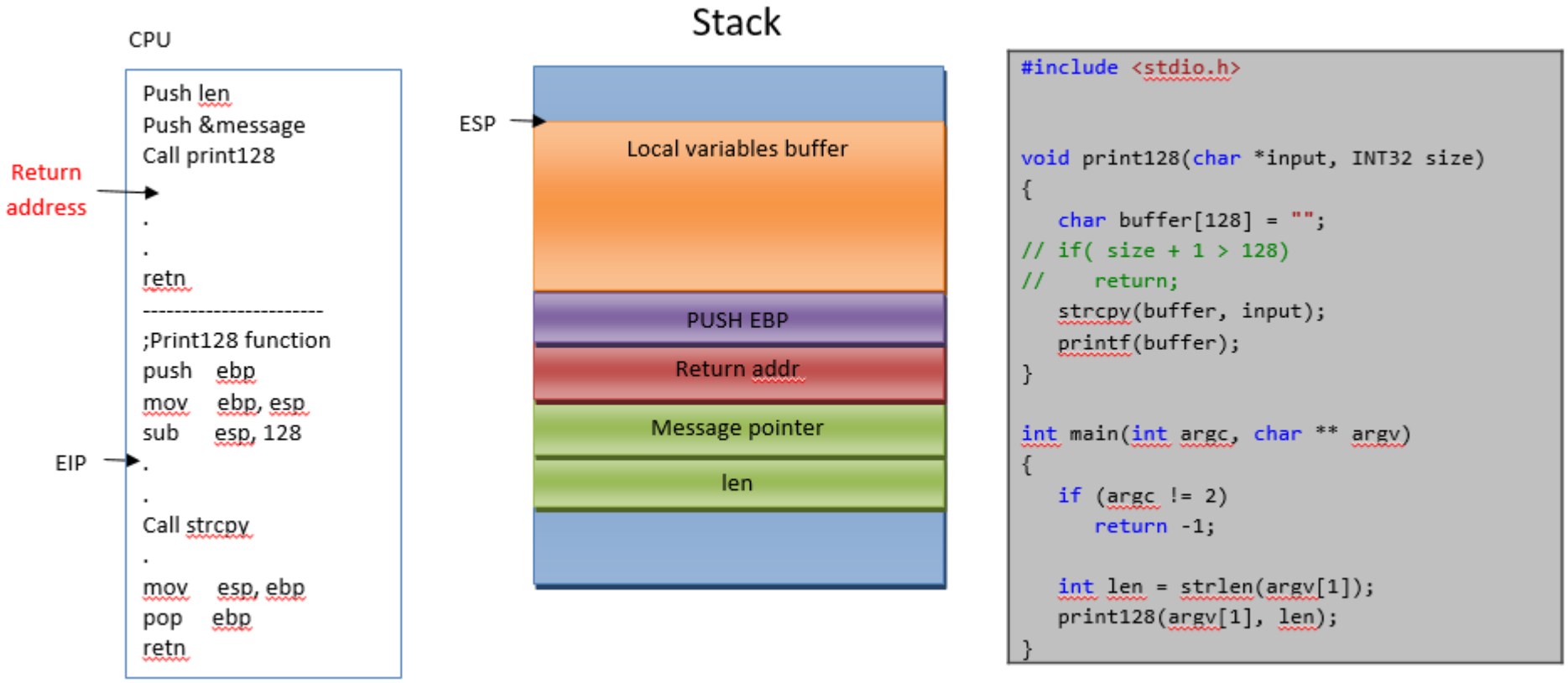
- Some protocol for functions to decide how to pass and return parameters and return value.
- Some conventions:
  - __cdecl (Default c language calling convention pushing parameters right-to-left on the stack)
  - __stdcall (Used in win32, same as cdecl but the responsibility of clearing parameters from stack is to the callee function)
  - __fastcall (Pass parameters by general purpose registers instead of stack)
  - __thiscall (Object oriented c++ codes passing this object in ecx register)

Stack overflow

- Local variable can store user input data
- If size of user input data is not checked the user can store more data to local variable buffer
- Attacker is able to overwrite memory of other variables on the stack and return address
- Overwriting return address lead to gaining arbitrary EIP value.
- So it is possible to control EIP (so CPU) to execute user input data as shellcode

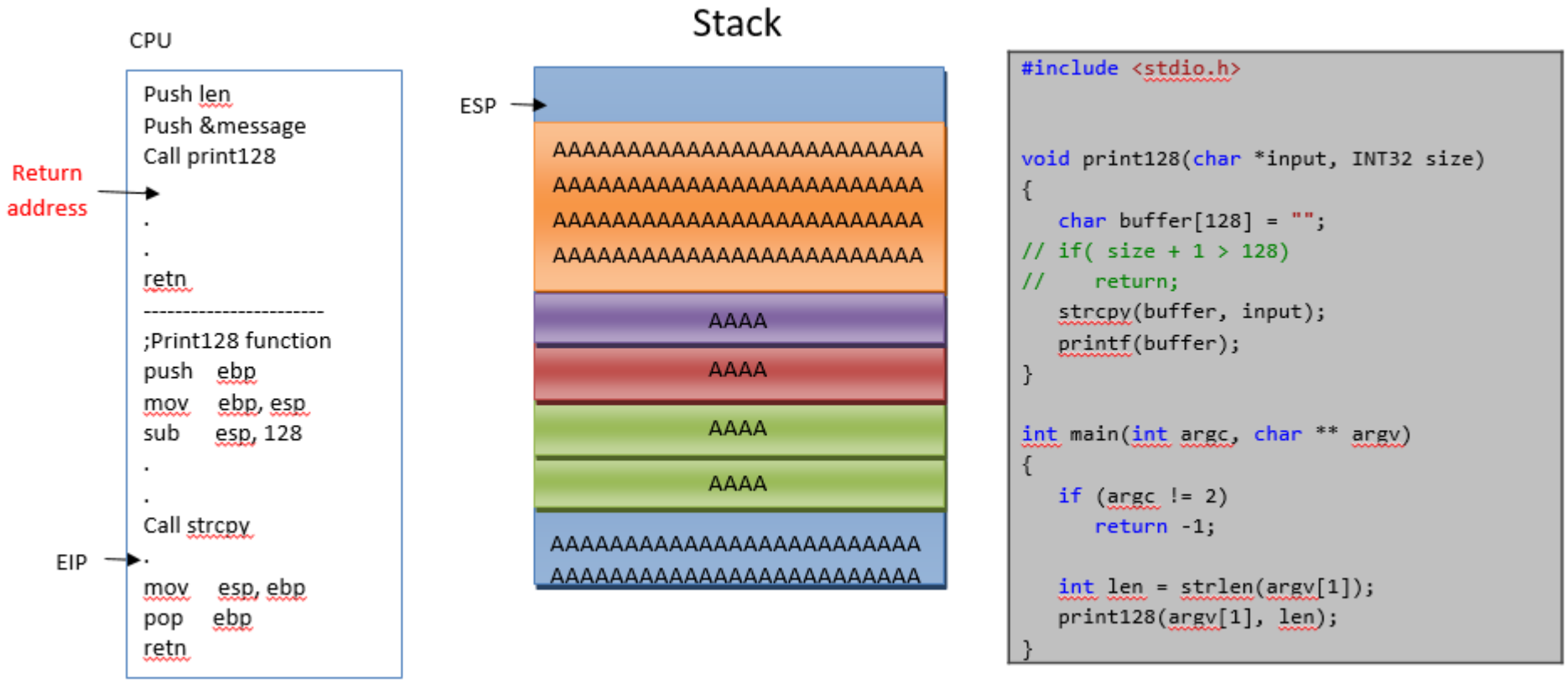


Stack Overflow – Cont'd



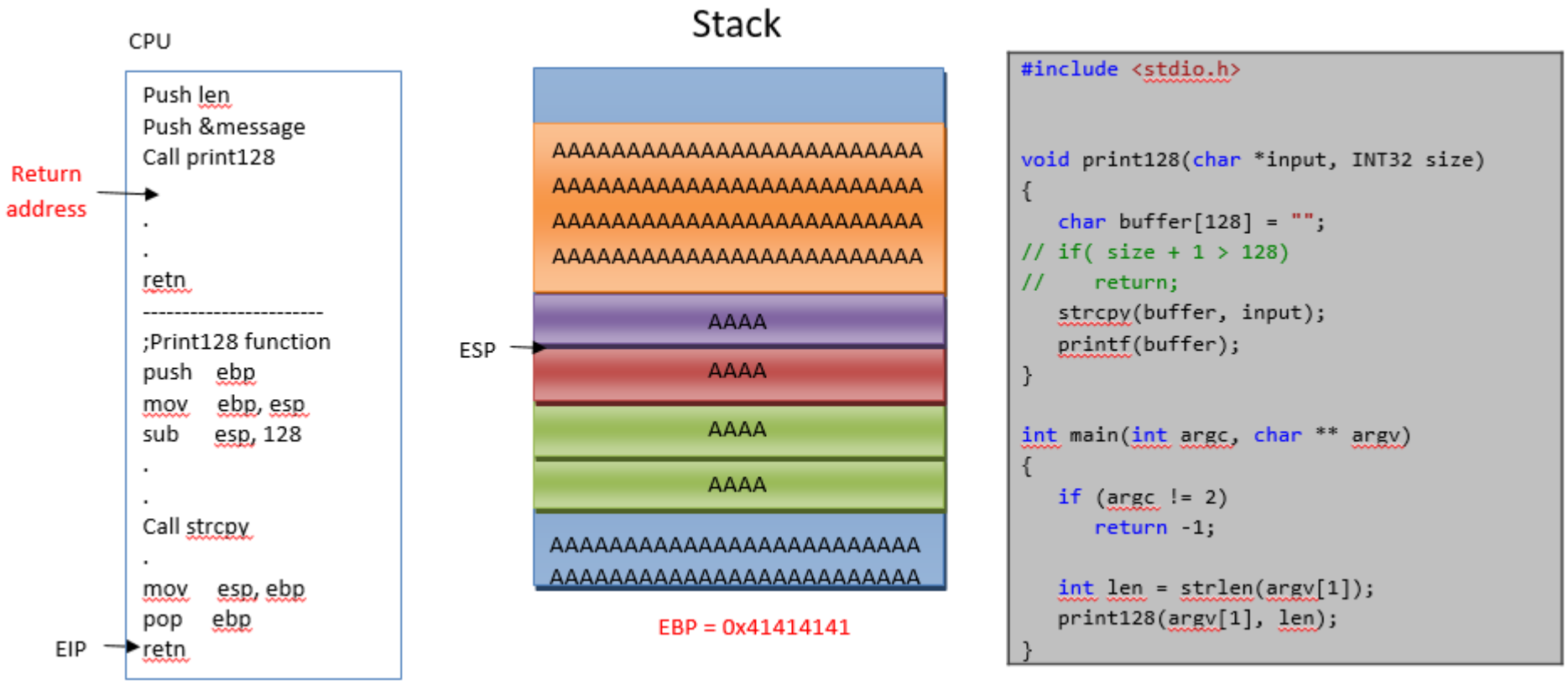

```c
#include <stdio.h>


void print128(char *input, INT32 size)
{
   char buffer[128] = "";
// if( size + 1 > 128)
//    return;
   strcpy(buffer, input);
   printf(buffer);
}

int main(int argc, char ** argv)
{
   if (argc != 2)
      return -1;

   int len = strlen(argv[1]);
   print128(argv[1], len);
}
```

Stack Overflow – Cont'd



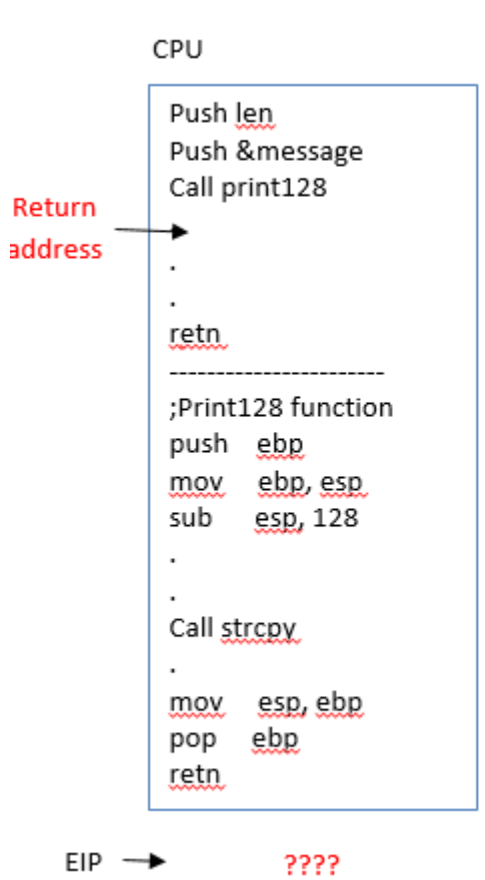

```c
#include <stdio.h>


void print128(char *input, INT32 size)
{
   char buffer[128] = "";
// if( size + 1 > 128)
//    return;
   strcpy(buffer, input);
   printf(buffer);
}

int main(int argc, char ** argv)
{
   if (argc != 2)
      return -1;

   int len = strlen(argv[1]);
   print128(argv[1], len);
}
```

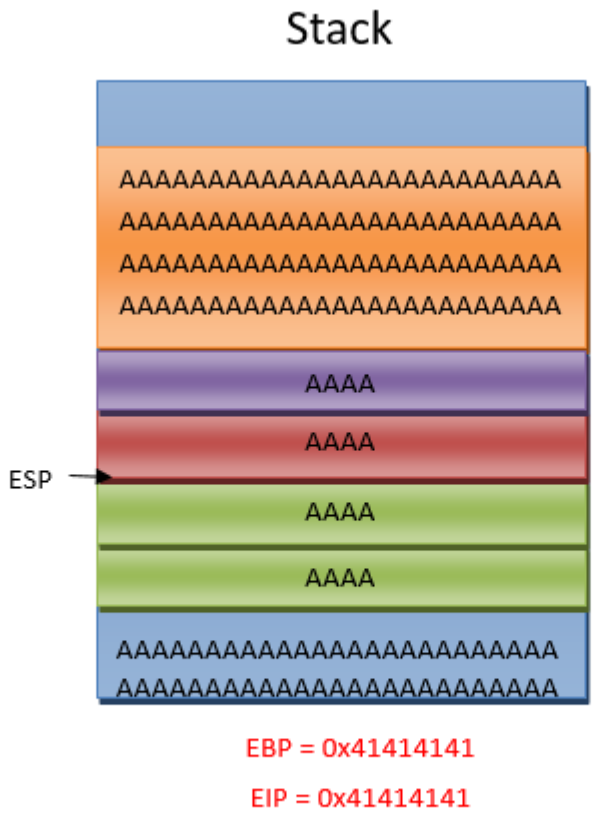
Stack Overflow – Cont'd




```c
#include <stdio.h>


void print128(char *input, INT32 size)
{
   char buffer[128] = "";
// if( size + 1 > 128)
//    return;
   strcpy(buffer, input);
   printf(buffer);
}

int main(int argc, char ** argv)
{
   if (argc != 2)
      return -1;

   int len = strlen(argv[1]);
   print128(argv[1], len);
}
```

Stack Overflow – Cont'd






```c
#include <stdio.h>


void print128(char *input, INT32 size)
{
   char buffer[128] = "";
// if( size + 1 > 128)
//    return;
   strcpy(buffer, input);
   printf(buffer);
}

int main(int argc, char ** argv)
{
   if (argc != 2)
      return -1;

   int len = strlen(argv[1]);
   print128(argv[1], len);
}
```

Stack Overflow – Cont'd

- The attacker can put any address to EIP register
- EIP register is the register x86 CPU uses to fetch and execute next instruction
- Shellcode can be delivered through the same user input that caused overflow or any other user input.
- Attacker put address of user data as shellcode in to the EPI register and execute malicious code.

Memory Protection (Exploit mitigation)

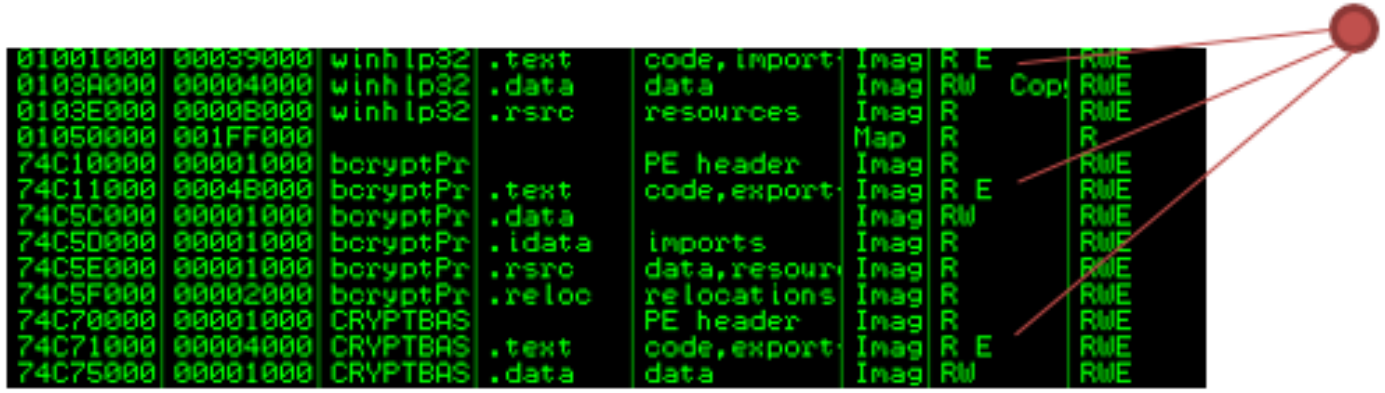
- Stop the attacker to gain code execution via memory corruption attacks
- In terms of action
  - Stop the attacker from gaining control over EIP (Stack Cookie)
  - Stop the attacker from finding shellcode address (ASLR)
  - Stop the attacker from executing shellcode (DEP)
  - …
- In terms of design
  - Operating System memory manager
  - Compiler code generation
  - Memory inspection
  - …

DEP (Data Execution Prevention)

- I got control over EIP, Does that mean I can execute shellcode? Unfortunately not.
- The processor don’t allow any code to get executed in any memory sections other than code segment by default(.text .code)
- You application is not allowed to execute any code from Stack, Heap, Data section

DEP (Data Execution Prevention)

- You can compile your application without the support of this security feature.
- But you can still allocate some memory page and set it flags as executable
  - Windows: VritualAlloc
  - *nix: mmap
- You can still change permission of a previously allocated memory region
  - Windows: Virtualprotect
  - *nix: mprotect



ROP

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.

```
mov esi, eax
pop ebp
ret
```

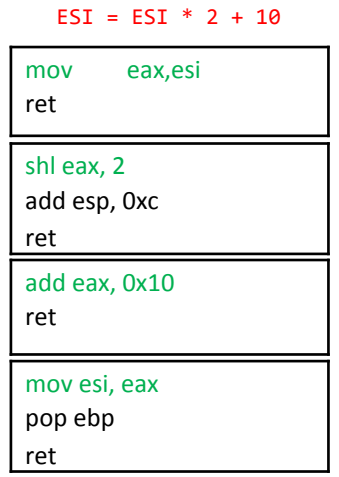
```
mov     eax,esi
ret
```

```
shl eax, 2
add esp, 0xc
ret
```

```
add eax, 0x10
ret
```

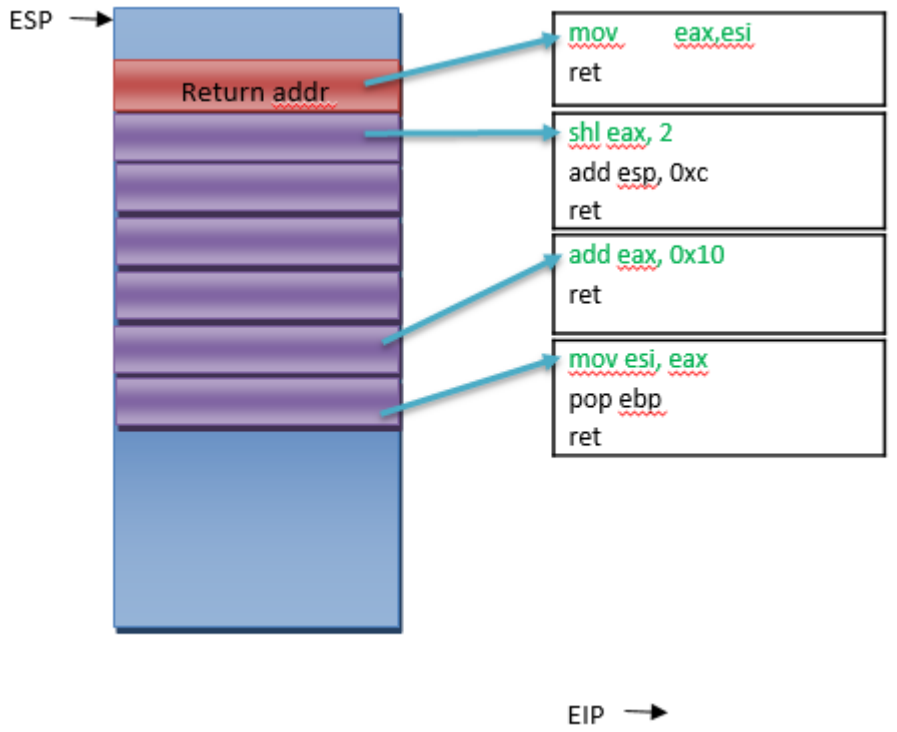
ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*



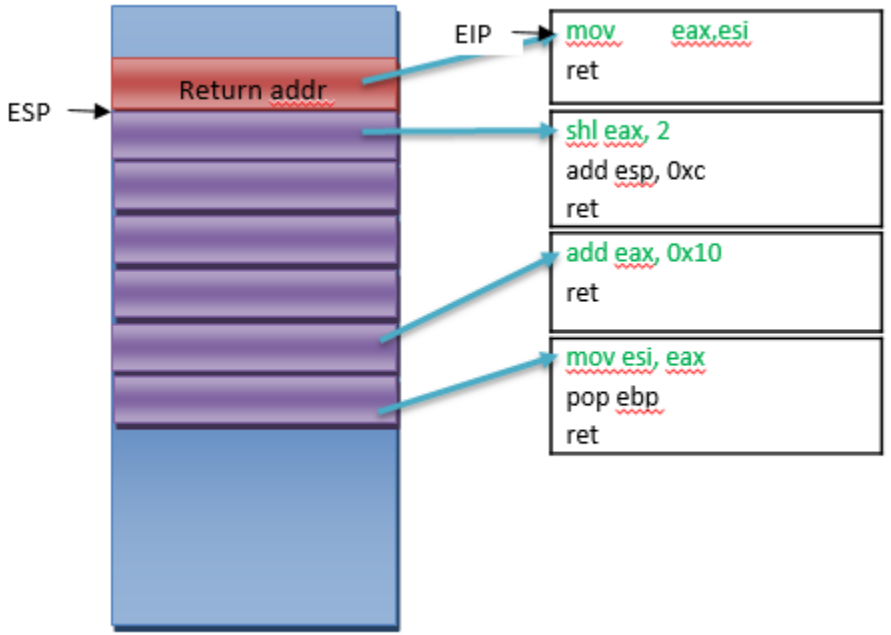
ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*
- There is a stack overflow? Attacker is able to put address of gadgets on the stack instead of real shellcode.



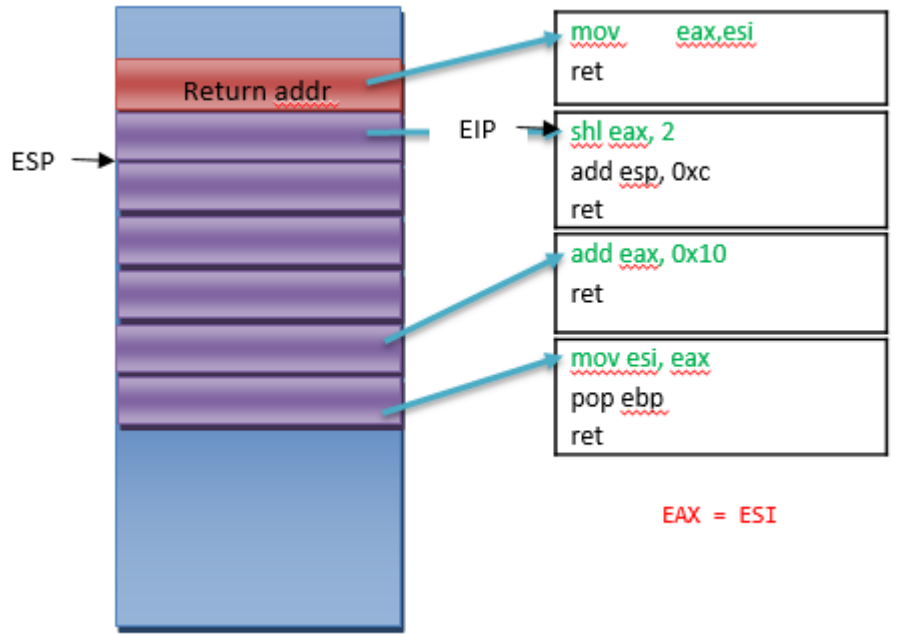
ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*
- There is a stack overflow? Attacker is able to put address of gadgets on the stack instead of real shellcode.



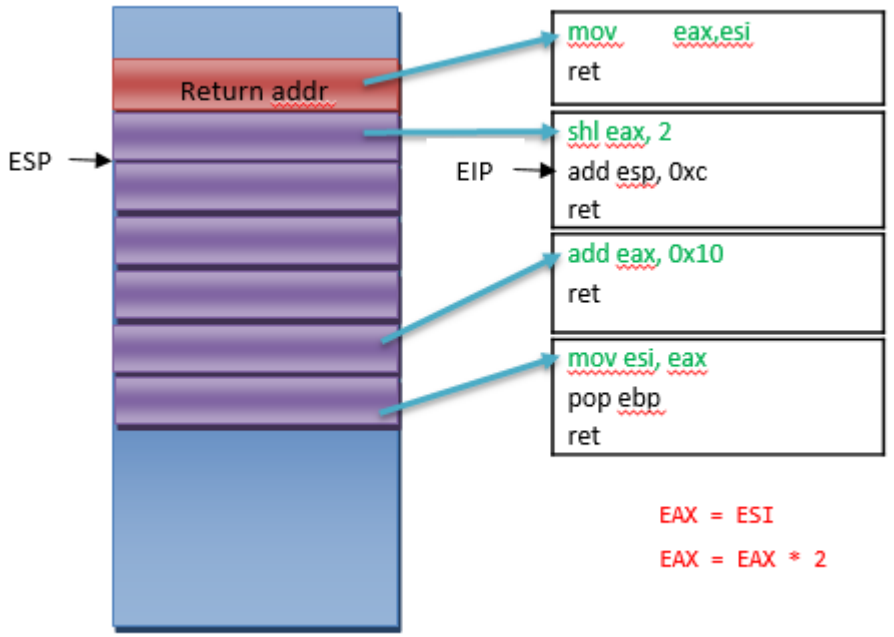
ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*
- There is a stack overflow? Attacker is able to put address of gadgets on the stack instead of real shellcode.



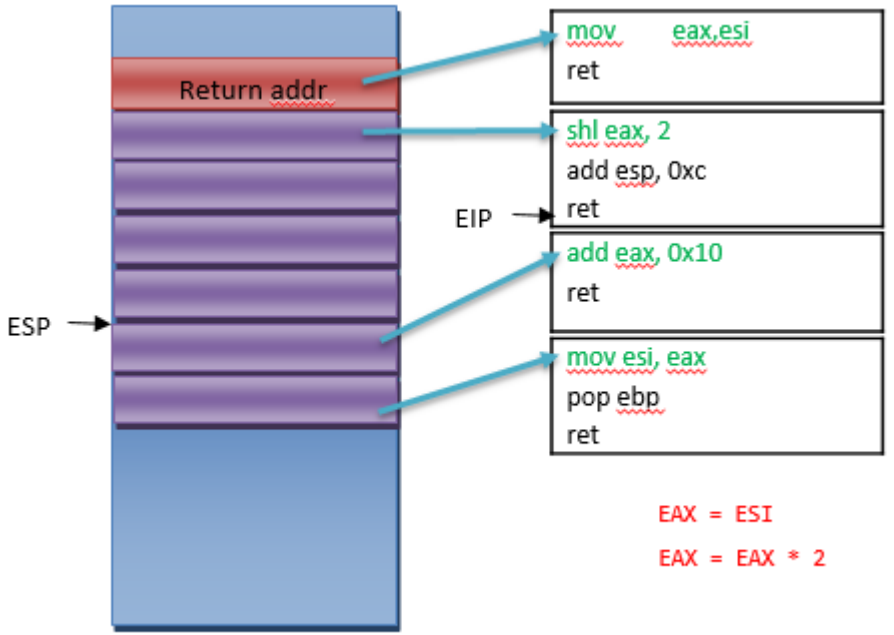
ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*
- There is a stack overflow? Attacker is able to put address of gadgets on the stack instead of real shellcode.



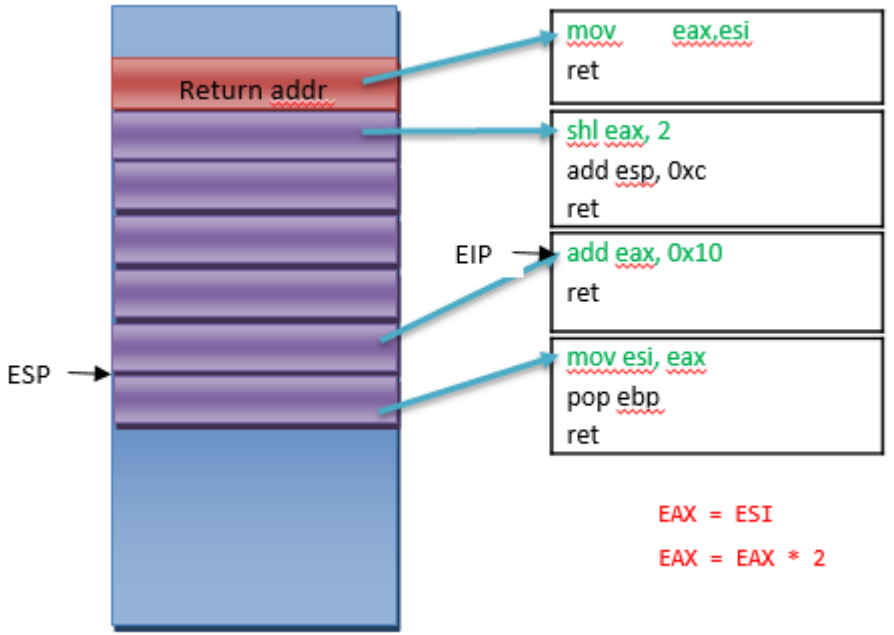
ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*
- There is a stack overflow? Attacker is able to put address of gadgets on the stack instead of real shellcode.



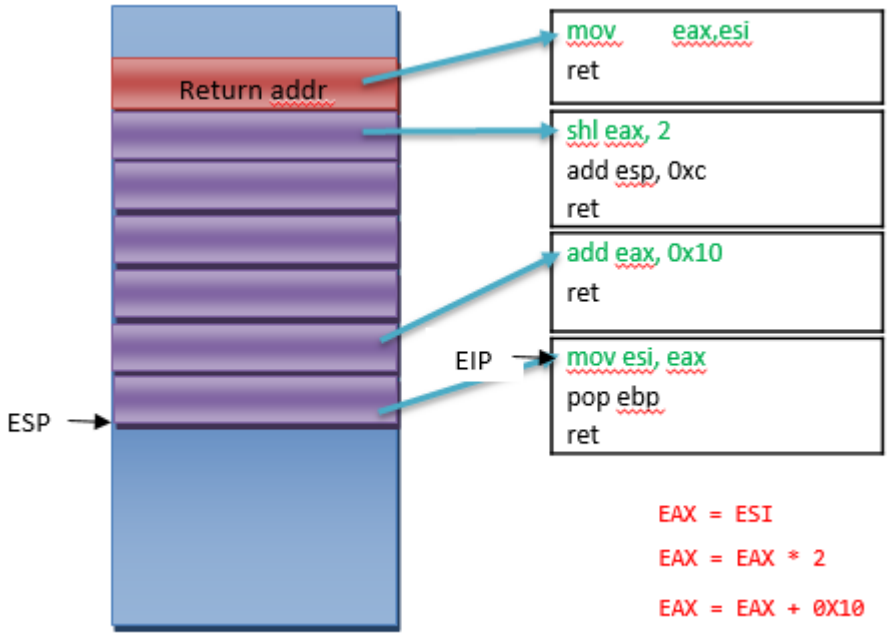
ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*
- There is a stack overflow? Attacker is able to put address of gadgets on the stack instead of real shellcode.



ROP – Cont'd

- There are some code sections mapped binary with RWX Permission.
- There are some *ret* instructions in the code
- A group of instructions that ends with *ret* is called gadget.
- If we arrange these gadgets together, we can make *meaningful code using existing code.*
- There is a stack overflow? Attacker is able to put address of gadgets on the stack instead of real shellcode.



ROP – Cont’d

- It is possible to write the whole shellcode using ROP Gadgets.
- As a trivial way, most of the time attackers uses small number of gadgets to execute the real shellcode
- Some examples are ROP Payloads that execute the following functions in windows:
  - VirtualAlloc + Memcpy + ret
  - Virtualprotect + ret
  - LoadLibrary (UNC path)

JOP – Jump Oriented Programming

- Similar to ROP attack but use *JMP/CALL* instructions instead of *RET*
- It gives more flexibility when there is not enough proper ROP Gadgets
- Some old ROP mitigations can be bypassed using JOP gadgets.

JOP Sample

1. *(DWORD*)(buffer) = imgBase + 0xE1E86;
   // MOV EAX, [ESI+4] | ADD ESI, 4 | ?? | ??
   | ?? | ?? | CALL EAX
2. *(DWORD*)(buffer+4) = imgBase +
   0x10FBC;
   // MOV EAX, [ESI+58h] |
   MOV ECX, [EAX+8] | ?? | ?? | ?? | ?? | CALL ECX
3. *(DWORD*)(buffer+8) = objPointer + 0x14;
4. *(DWORD*)(buffer+0xc) = imgBase +
   0x73c2;
   // ADD ESP, 8h | RETN
5. *(DWORD*)(buffer+0x10) = imgBase +
   0xD80F;
   // XCHG EAX, ESP | ?? | ?? |
   POP ESI | RETN
6. *(DWORD*)(buffer+0x18) = imgBase + 0xE6F0A;
   // CALL LoadLibraryW
7. *(DWORD*)(buffer+0x1C) = objPointer + 0x60; // lpLibFileName

Another Code Reuse Attack Scenario

- Antivirus/Protection Software hook some API to see if an untrusted code call a sequence of API in a malicious flow
- It is possible to use Code Reuse attack on running processes and bypass detections

Some Proposed Mitigations

- Randomization: Randomize code in image level or instruction level to stop attacker from arranging predictable gadget sequence
- Control flow integrity (CFI): Create a control flow graph (CFG) from code and monitor any invalid chain of control flow instructions JMP/CALL/RET based on the graph.

Research Ideas?

It is a very old attack but still exist. Research ideas regarding this topic?