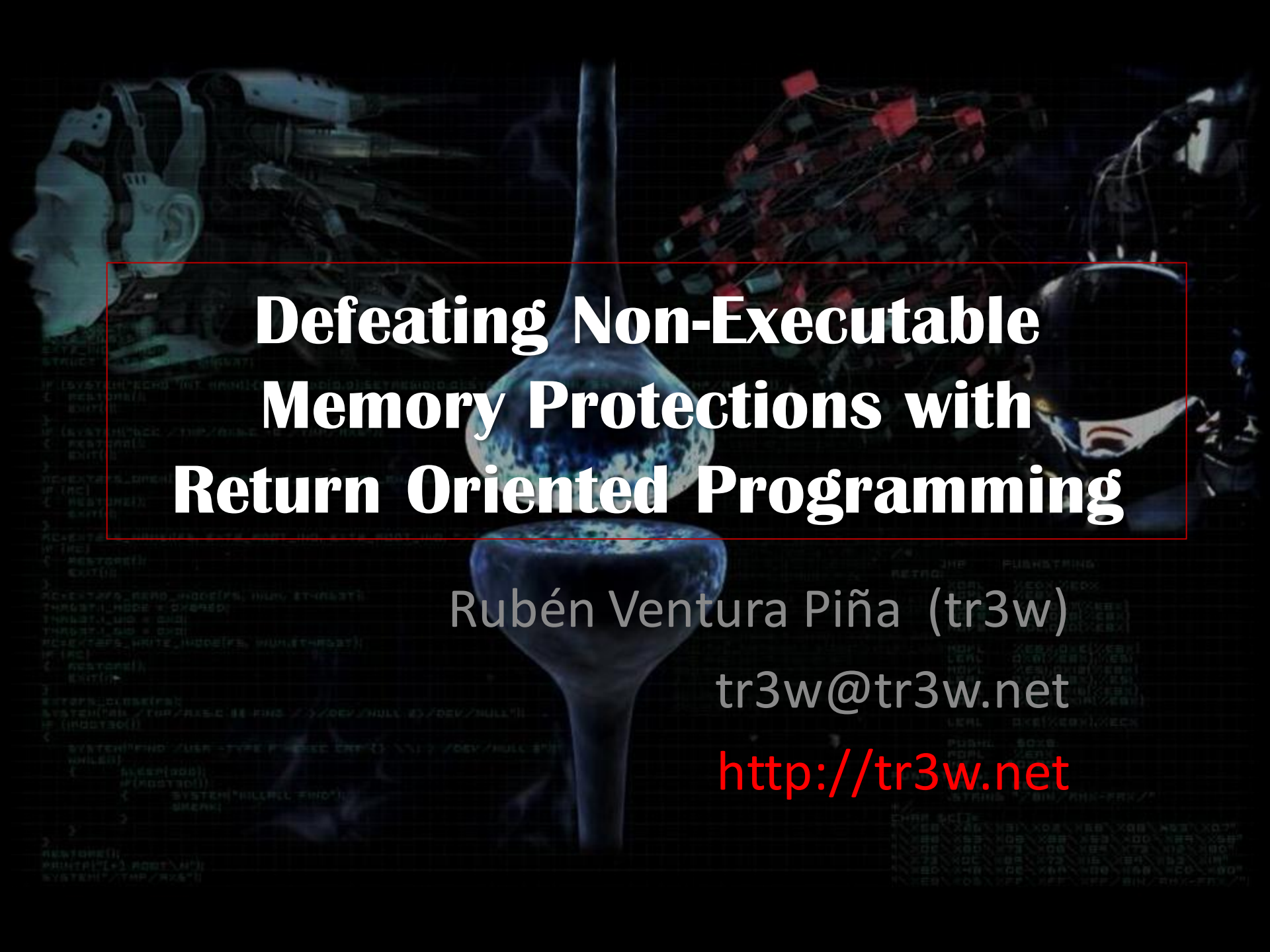


Defeating Non-Executable Memory Protections with Return Oriented Programming

Rubén Ventura Piña (tr3w)

tr3w@tr3w.net

http://tr3w.net

Agenda

- Intro
- Evolution of Memory Corruption Bugs
  - Exploitation techniques
  - Mitigations
  - Counter-Mitigations
- Return Oriented Programming
  - Analysis of Protection Mechanisms
  - Exploiting Windows
  - Exploiting Linux
- Affected technologies
- Outro

whoami

- Rubén Ventura Piña (tr3w)
  twitter: trew_0
- Has worked giving application security assessment
- Speaker at some cons
- Experience as a trainer

Memory Corruption Vulnerabilities

- One of the oldest types of attacks
- High risk
- They are still here
- Yet another defense mechanism, yet another bypass

Memory Corruption Vulnerabilities

- Vulnerability development becomes much more harder
- Exploitation techniques become much more complex
- EIP=0x41414141 is harder to get
- Writing a reliable exploit takes a lot of time.

Memory Corruption Vulnerabilities

- Compiler protection mechanisms
  - Stack Cookies
  - ‘Ideal Frame’ layout
  - Safe SEH
- OS protection mechanisms
  - Heap protections
  - NX Memory
  - ASLR

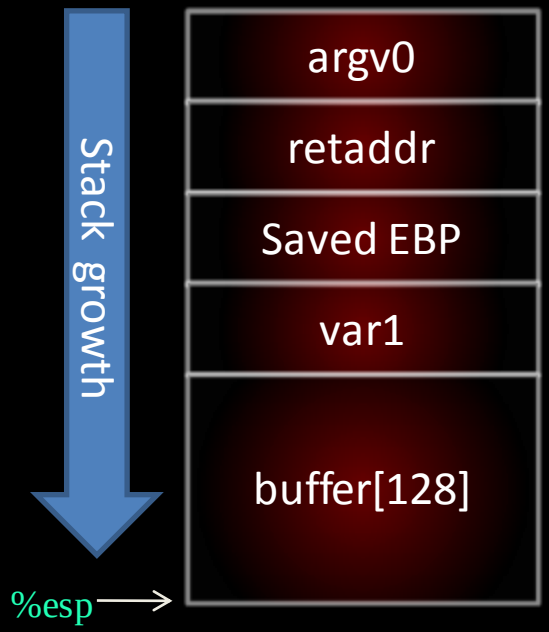
Memory Corruption Vulnerabilities

- Compiler protection mechanisms
  - Stack Cookies
  - 'Ideal Frame' layout
  - Safe SEH
- OS protection mechanisms
  - Heap protections
  - NX Memory
  - ASLR

Return Oriented Programming

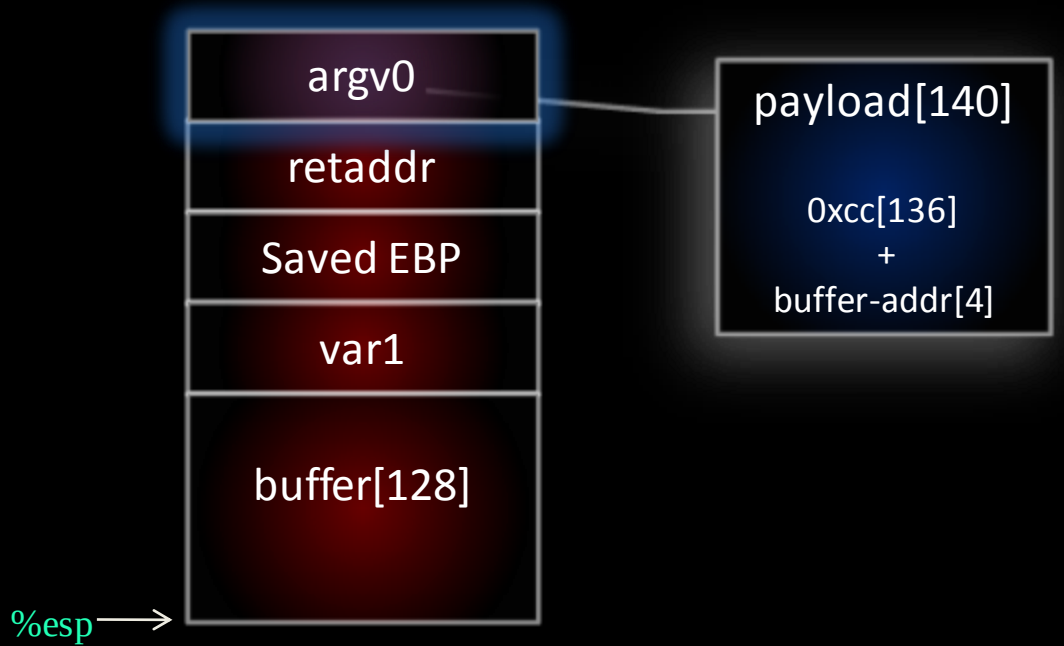
- Modern exploitation technique
- Bypass to actual protection mechanisms
- Getting very popular

Code Injection



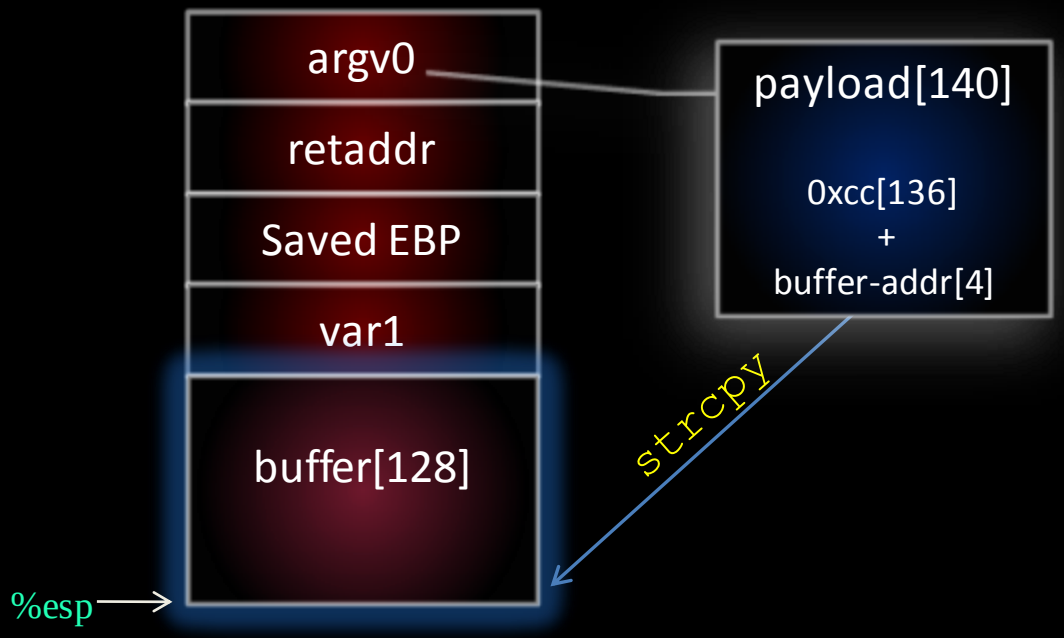

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

Code Injection



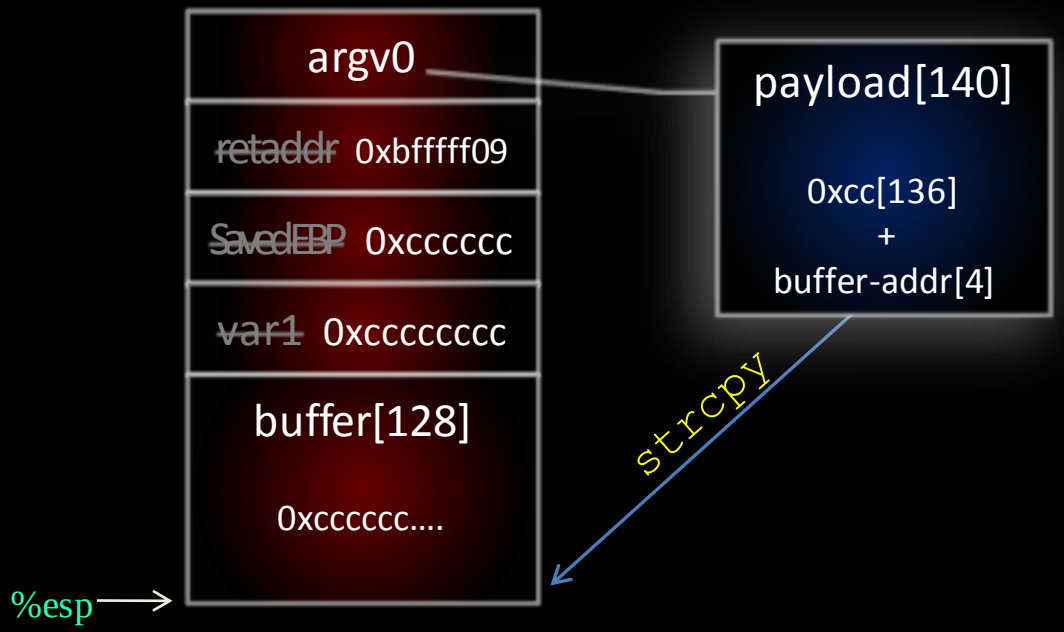

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

Code Injection




```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

Code Injection

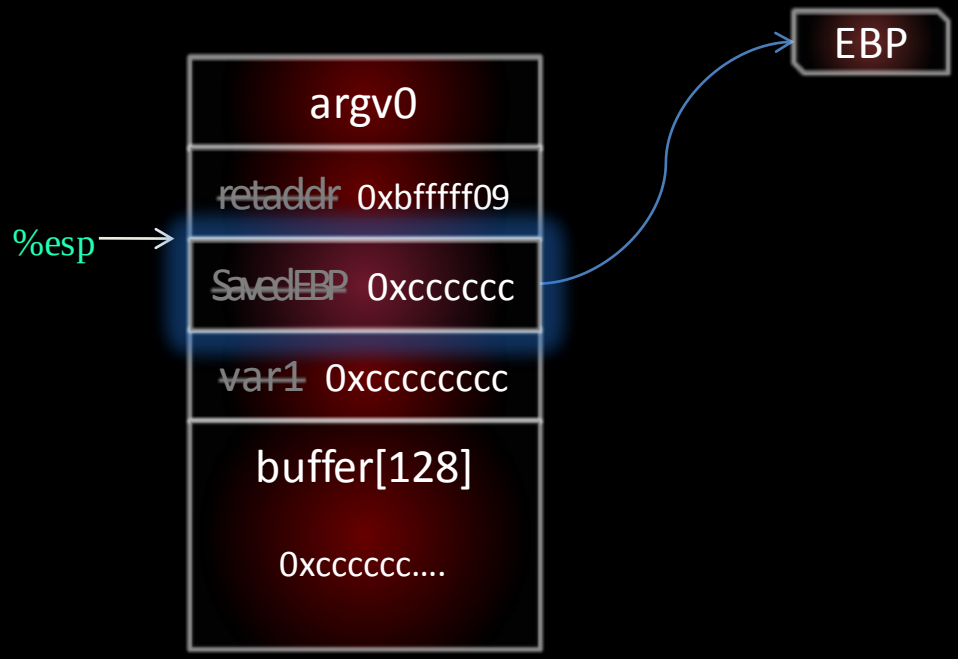



```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

Code Injection

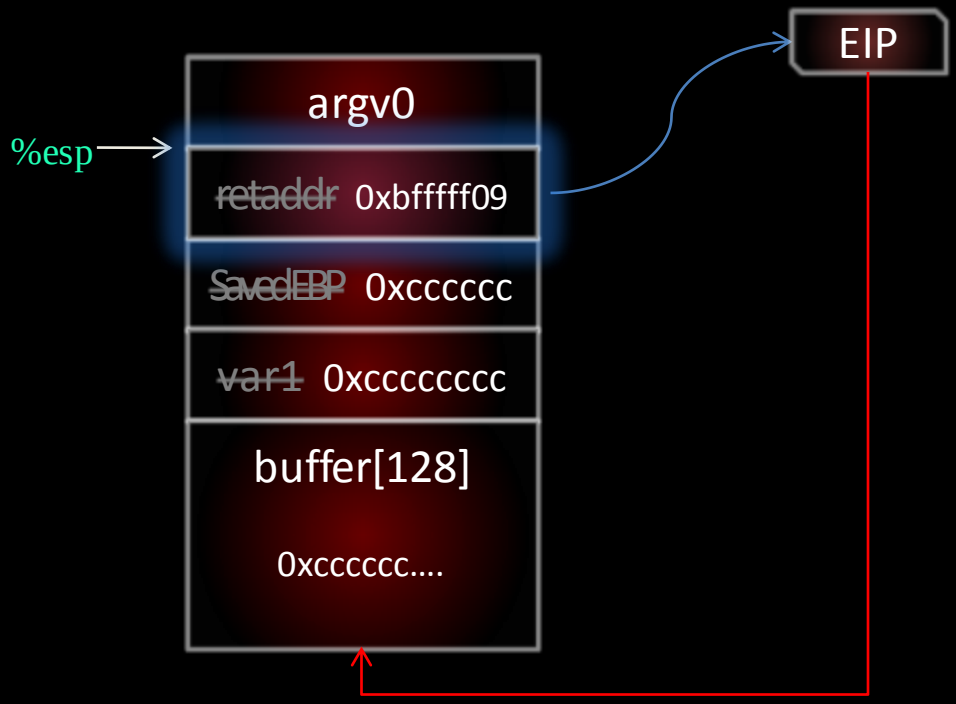

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

Code Injection



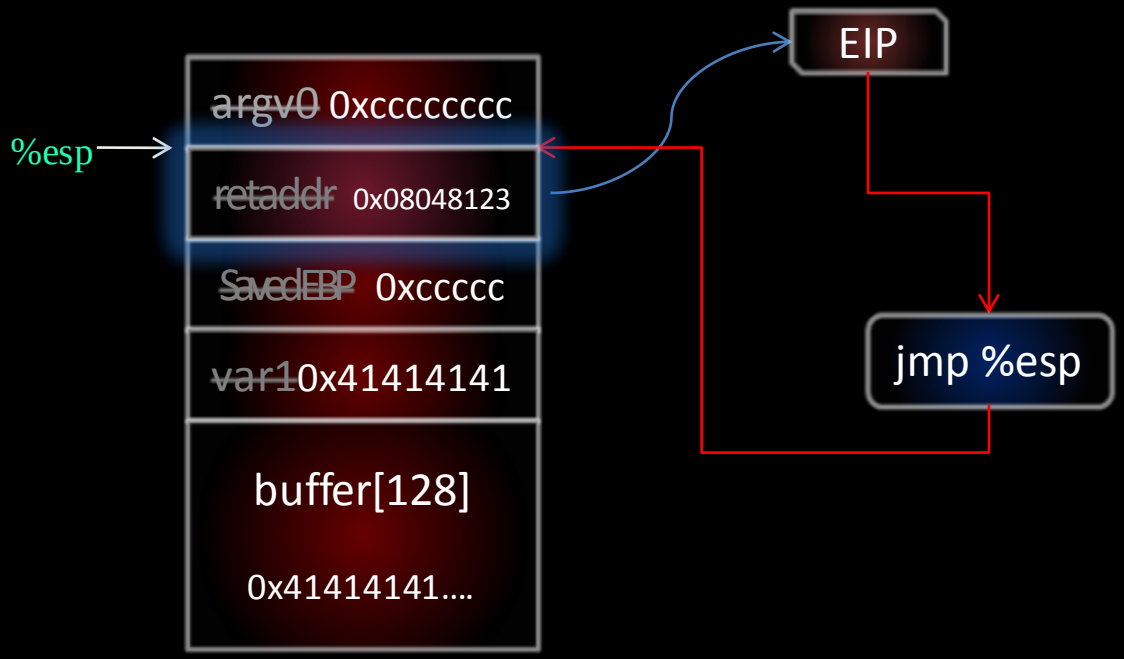

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

Code Injection



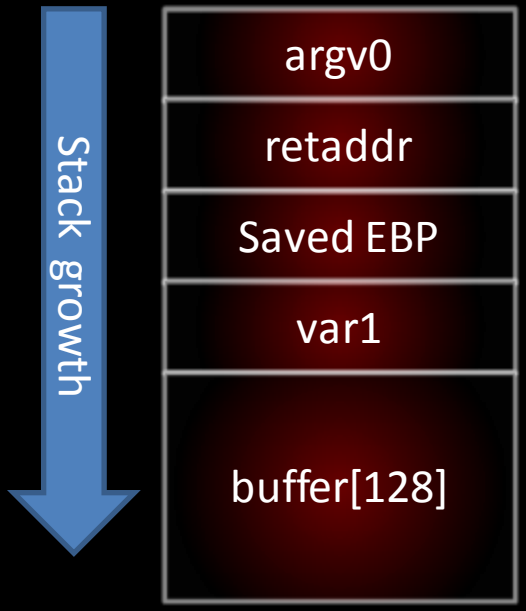

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

Code Injection: jmp-through-reg




```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

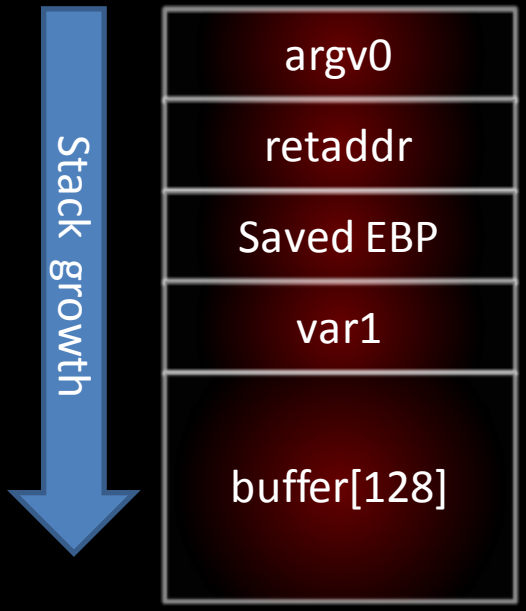
Code Injection



EIP is manipulated to redirect execution to the stack

Mitigations???

Code Injection



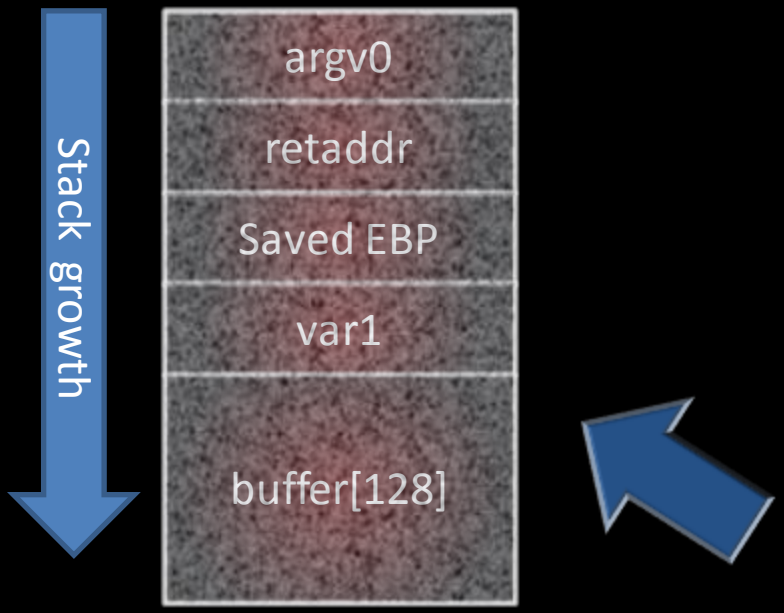
EIP is manipulated to redirect execution to the stack

Mitigations???

Non-Executable Stack

```
rwxp    -->    rw-p
```

Code Injection



EIP is manipulated to redirect execution to the stack

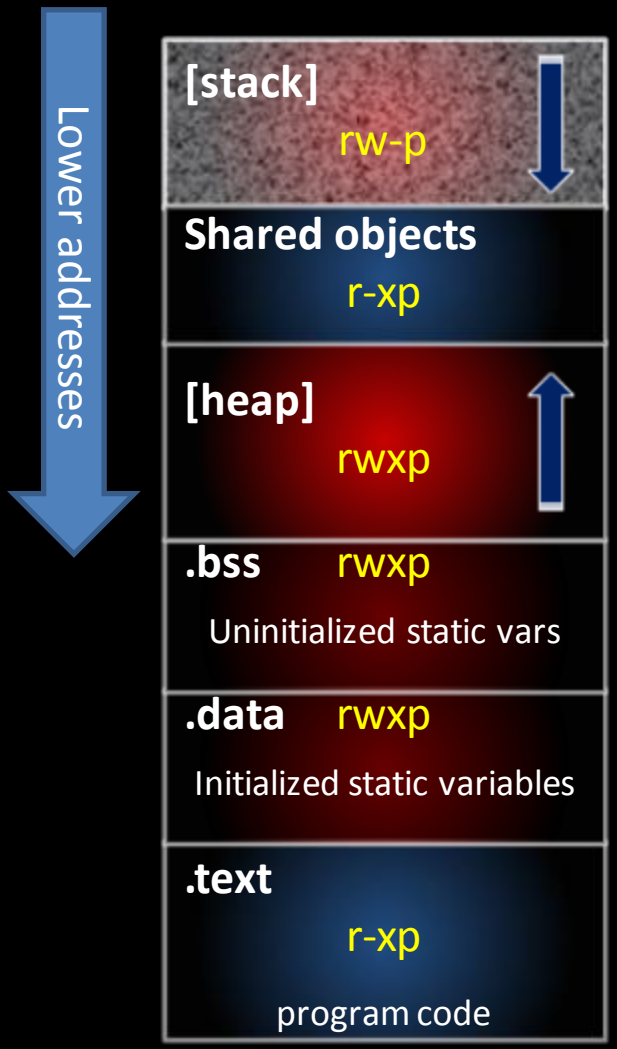
Mitigations???

Non-Executable Stack

`rwxp --> rw-p`

Placing shellcode in the stack and then jumping to it is not an option anymore

nx-stack



-> Linux: Solar Designer, 04-1997

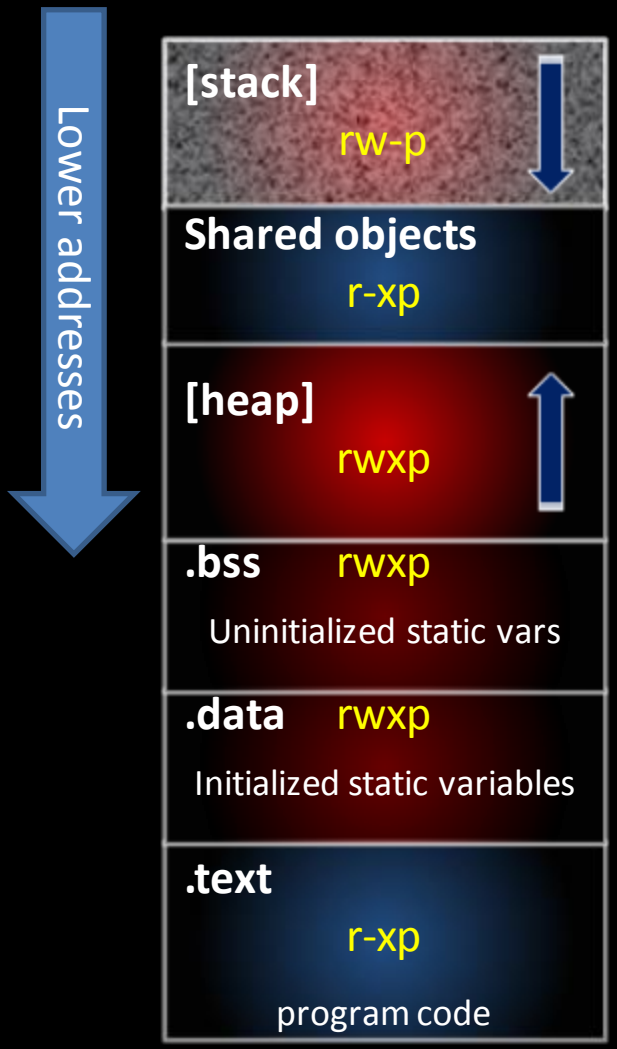
- Adopted in other OS as well.

Weakness?

Code can be executed everywhere else!

Place shellcode in any other section and jump to it.

nx-stack



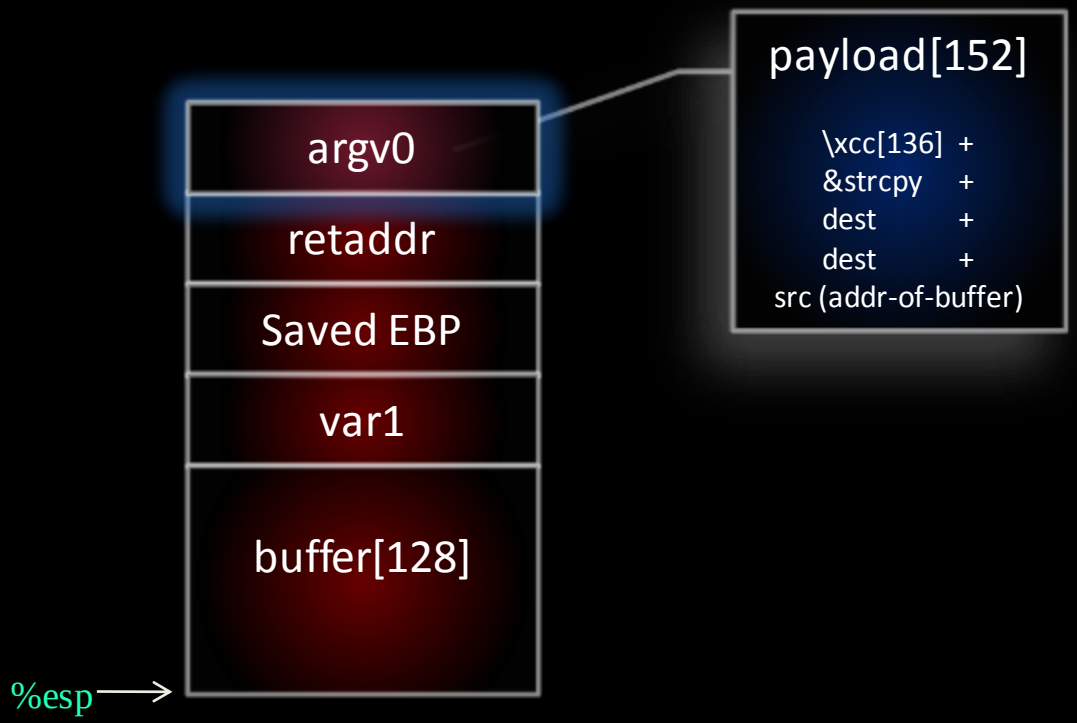
-> Linux: Solar Designer, 04-1997
- Adopted in other OS as well.

Weakness?

Code can be executed everywhere else!

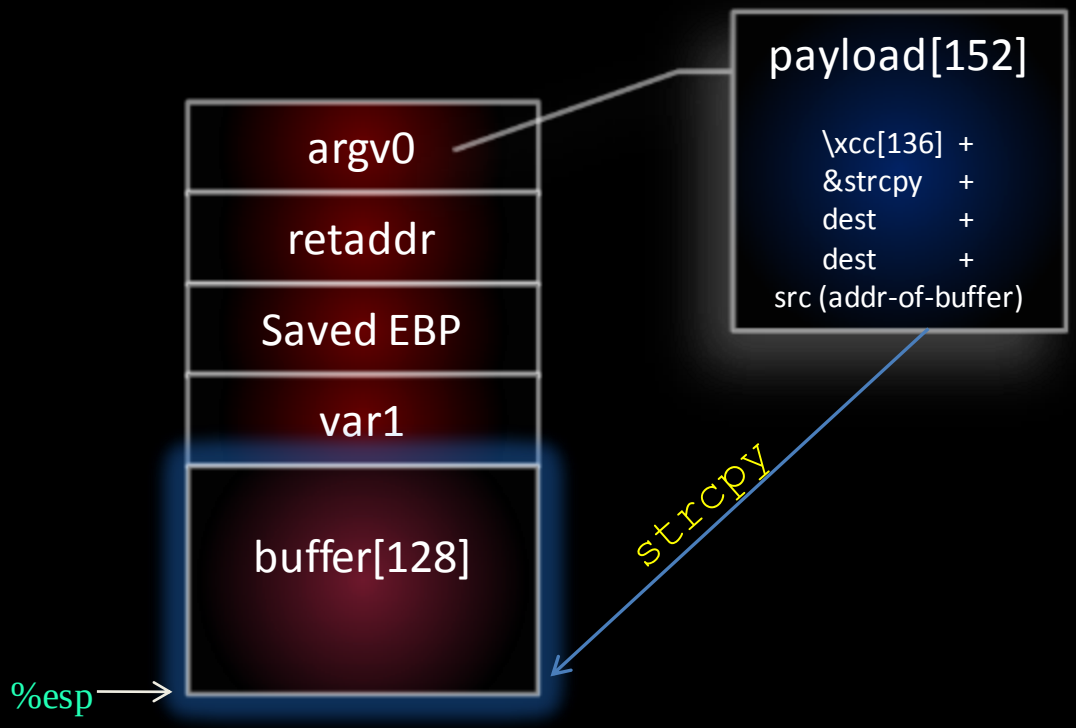
Place shellcode in any other section and jump to it.

ret2strcpy



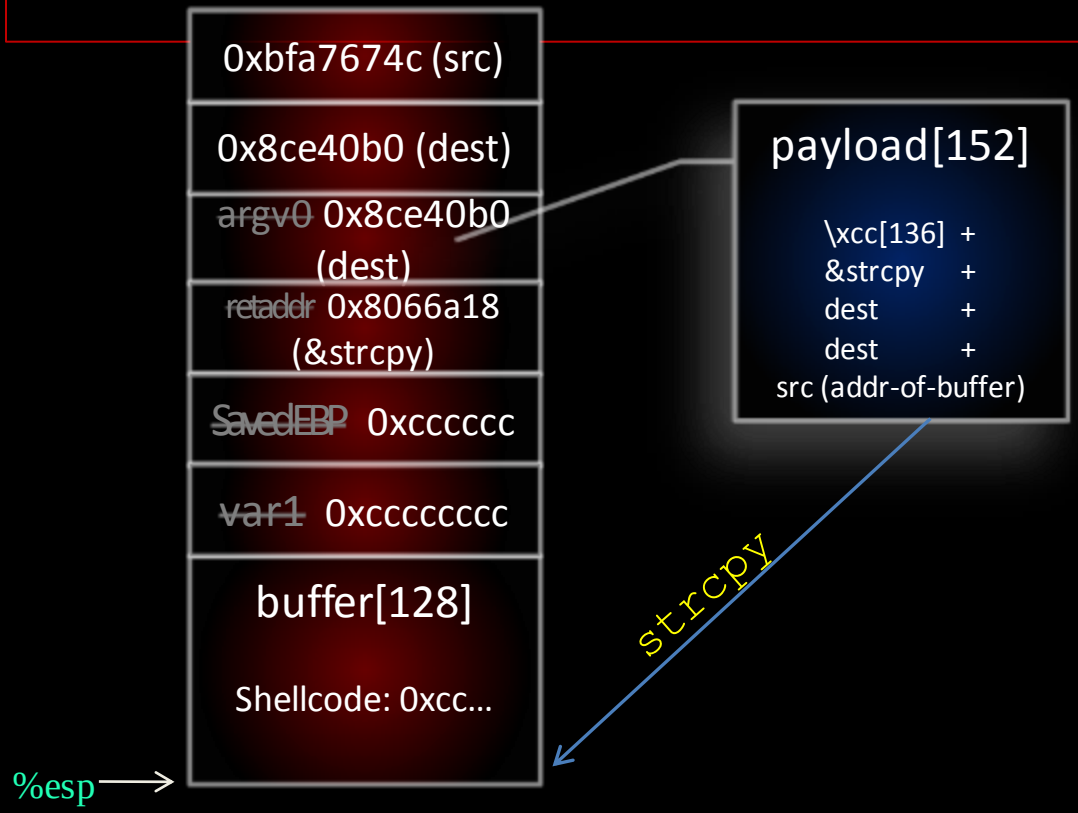

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy



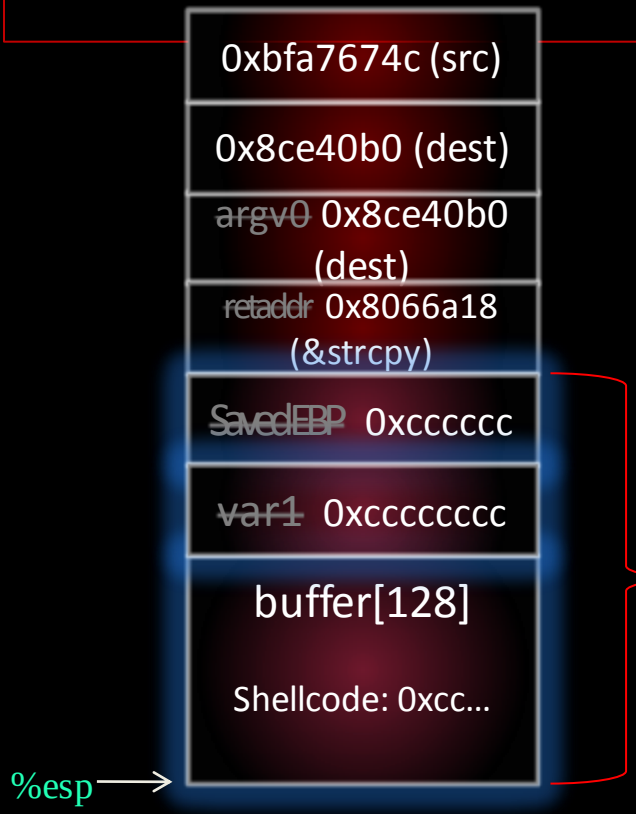

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy




```
int vulnerable(char
*argv0)
{
    int var1;
    char buffer[128];

    strcpy (buffer, argv0);
    return 0;
}
```

ret2strcpy



Shellcode (0xcc)

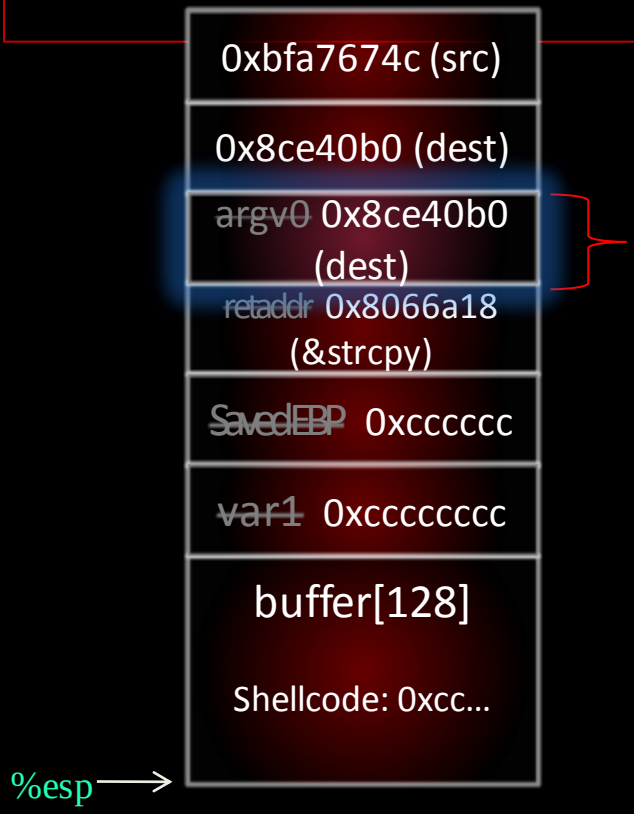
```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy

Address of strcpy() in libc

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy



Address of destination buffer (eg somewhere in the heap)

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy

Same as below

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy

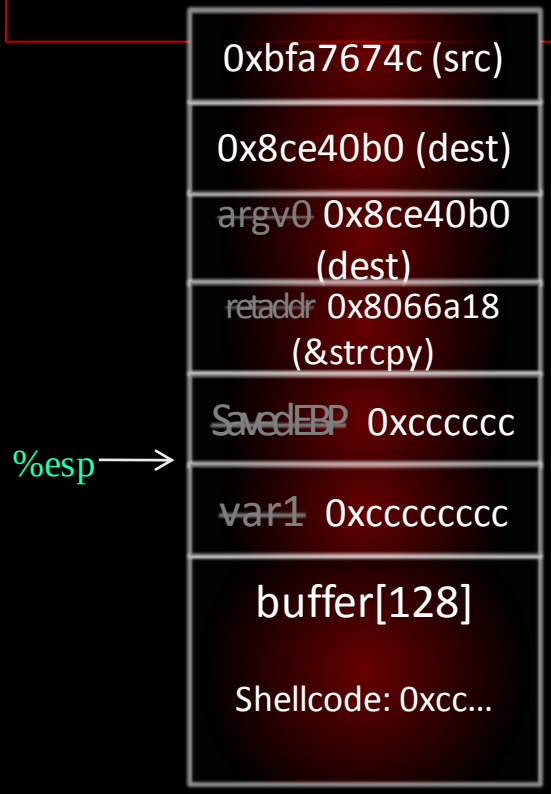
Address of source buffer
(Pointer to shellcode)

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy

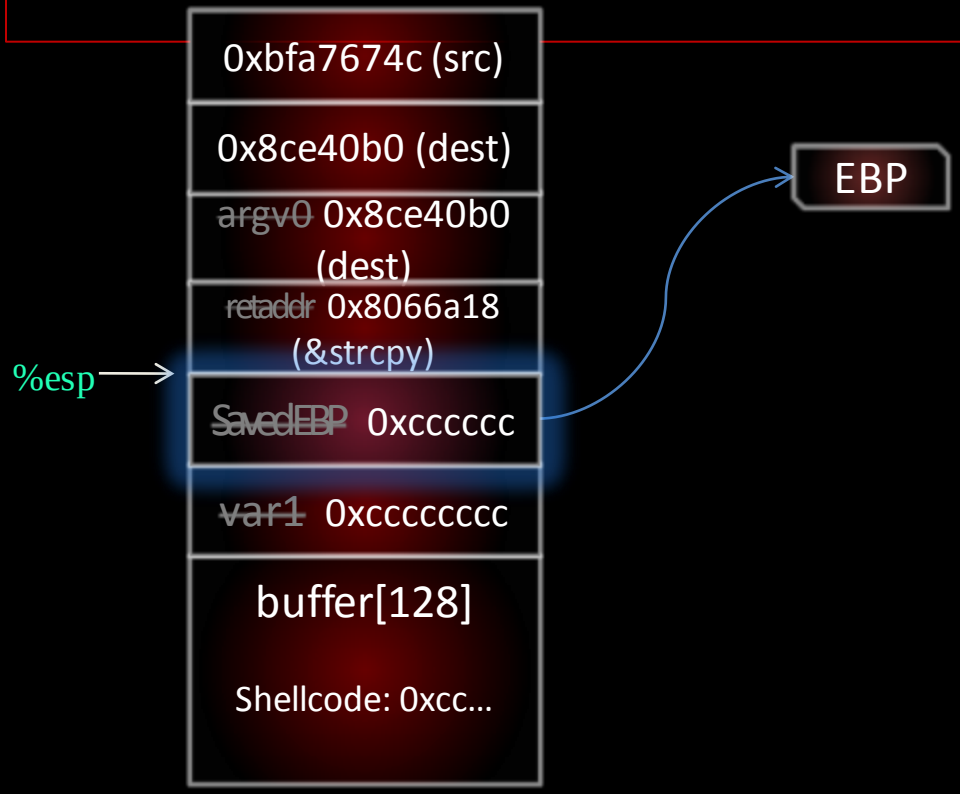

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}
```

ret2strcpy



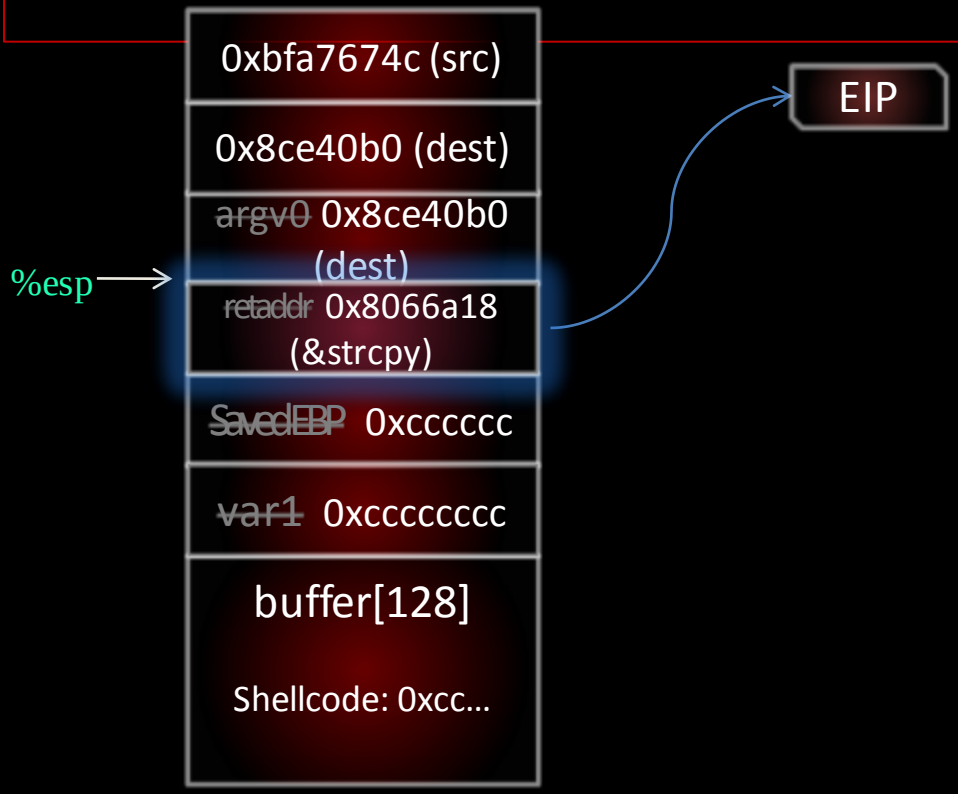

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

ret2strcpy



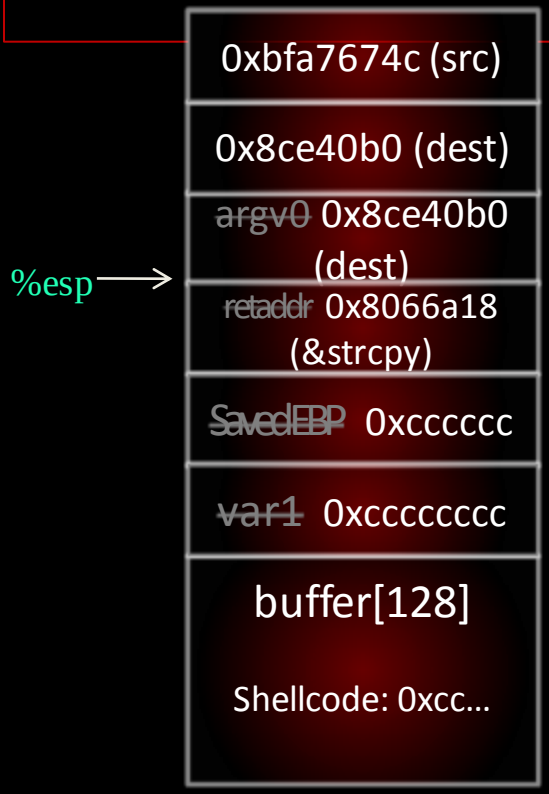

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

ret2strcpy




```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

ret2strcpy



EIP →

```
<strcpy>:
push    %ebp
mov     %esp, %ebp
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi

. . .

pop     %ebp
ret
```

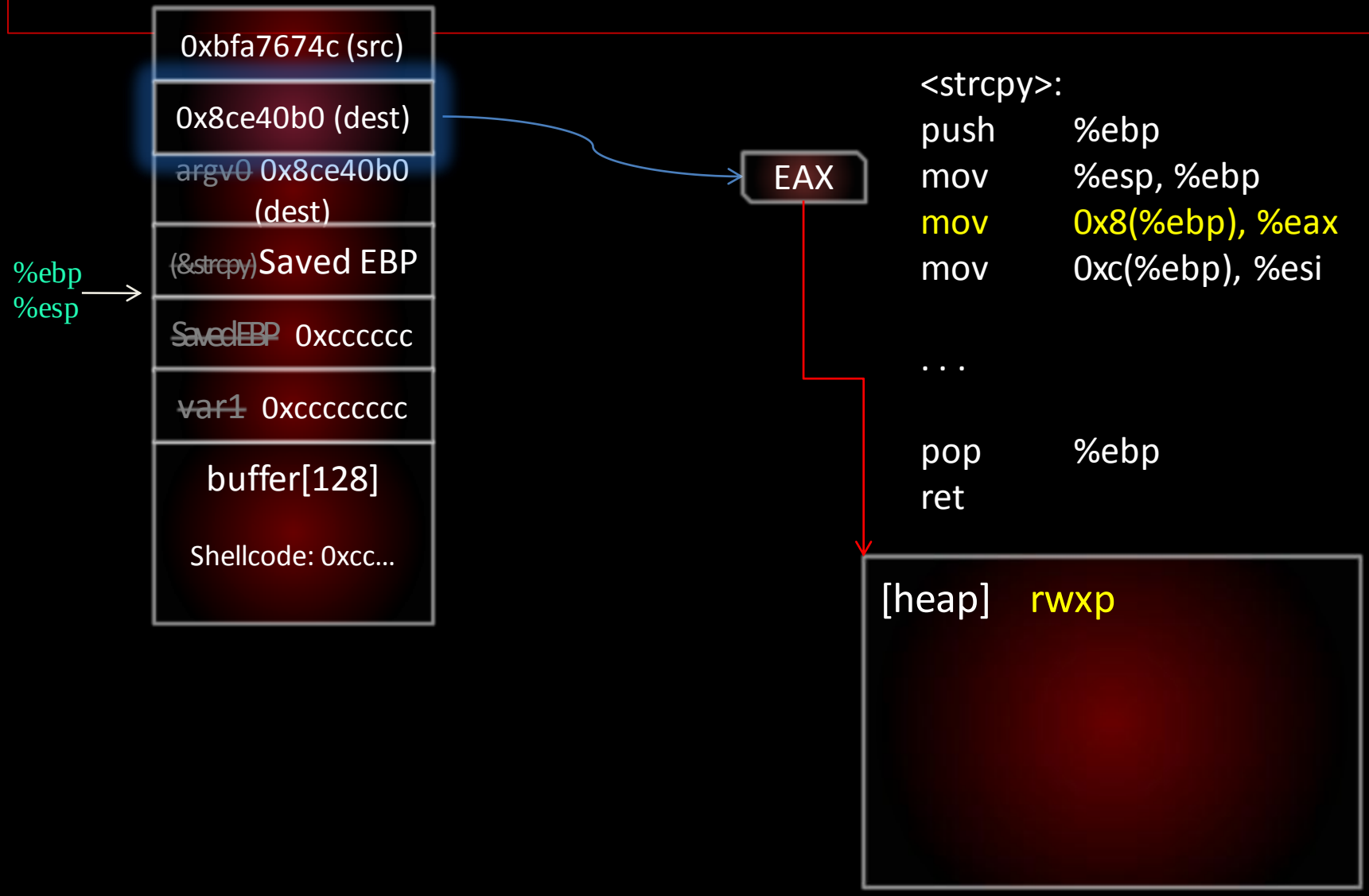
ret2strcpy


```
<strcpy>:
push    %ebp
mov     %esp, %ebp
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi

. . .

pop     %ebp
ret
```

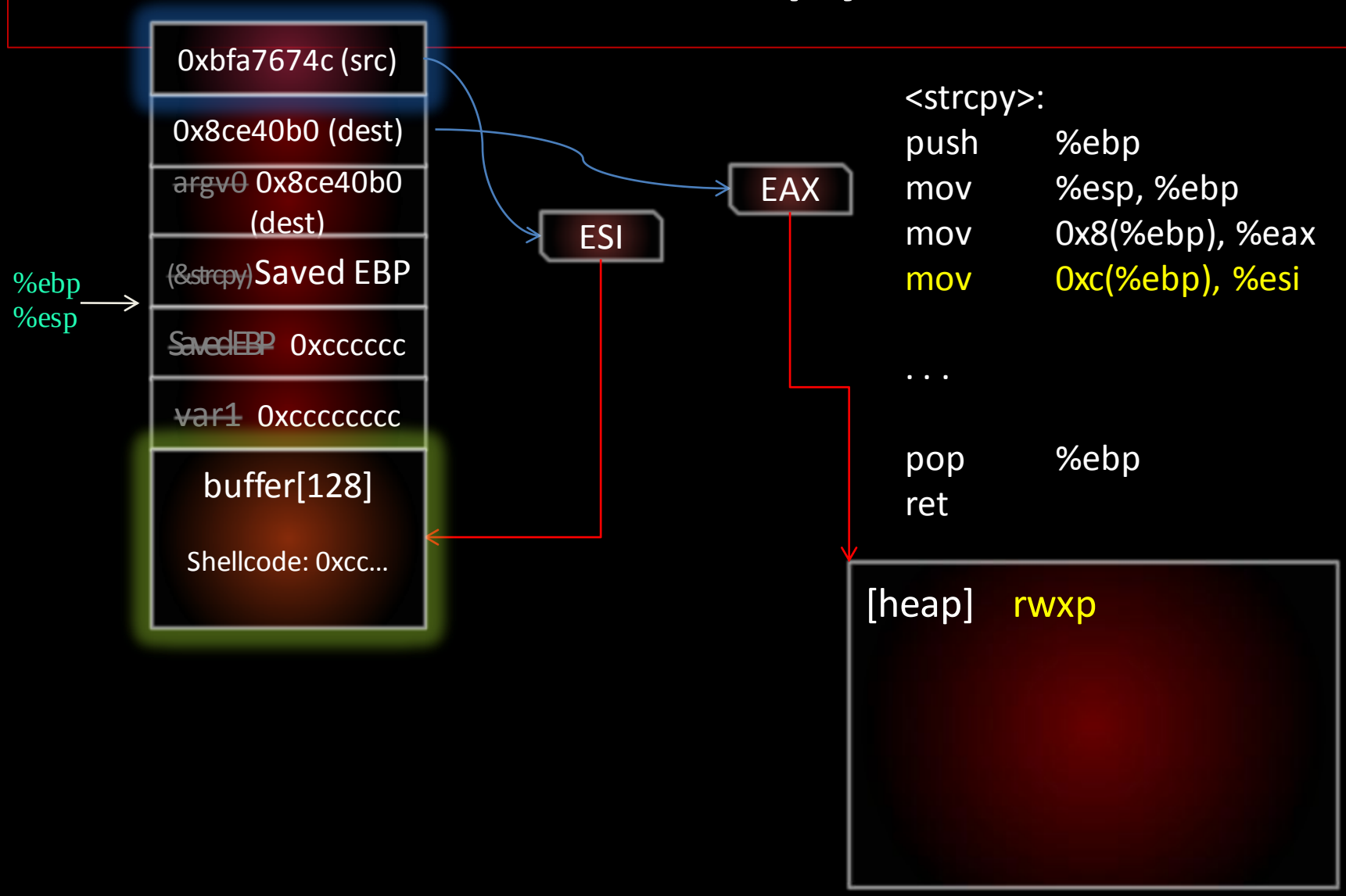
ret2strcpy


```
<strcpy>:
push    %ebp
mov     %esp, %ebp
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi

. . .

pop     %ebp
ret
```

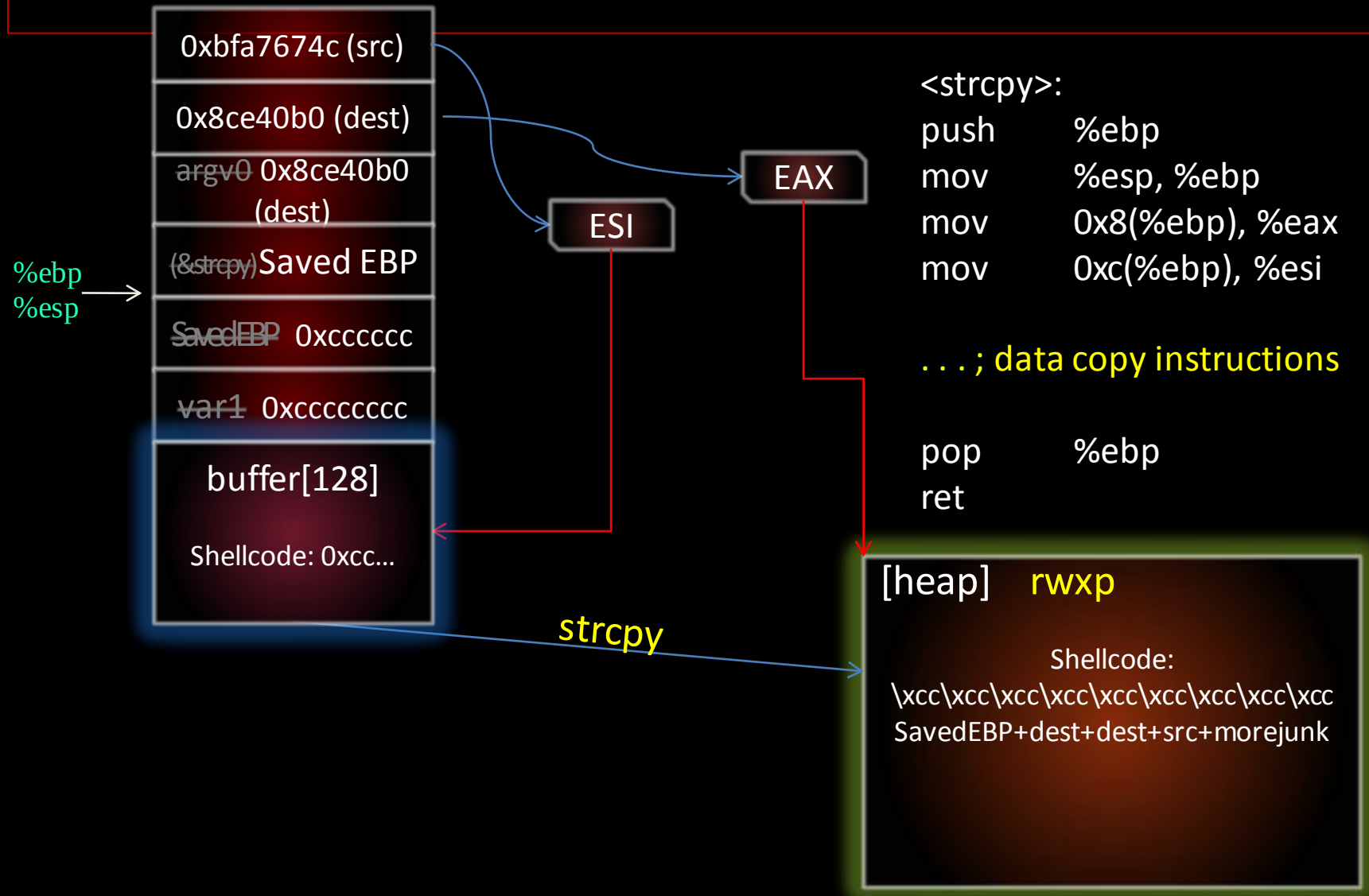
ret2strcpy


0xbfa7674c (src)
0x8ce40b0 (dest)
argv0 0x8ce40b0 (dest)
(&strcpy) Saved EBP
SavedEBP 0xcccccc
var1 0xcccccccc
buffer[128]
Shellcode: 0xcc...
%ebp
%esp
EAX
<strcpy>:
push %ebp
mov %esp, %ebp
mov 0x8(%ebp), %eax
mov 0xc(%ebp), %esi
. . .
pop %ebp
ret
[heap] rwxp

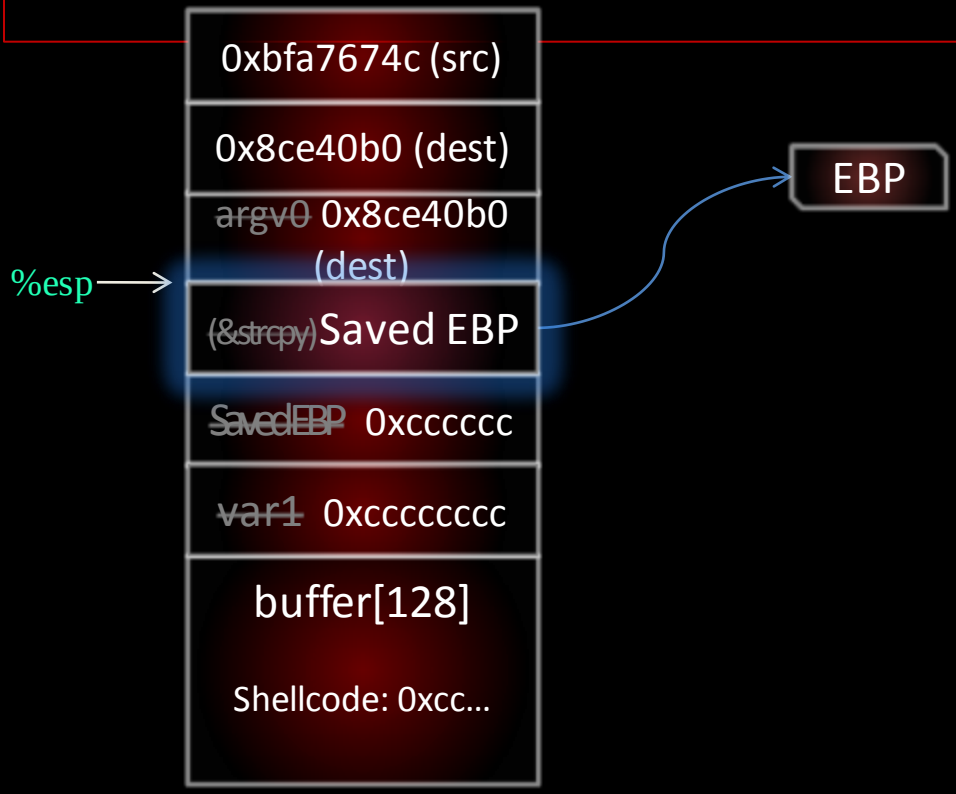
ret2strcpy


0xbfa7674c (src)
0x8ce40b0 (dest)
argv0 0x8ce40b0 (dest)
(&strcpy) Saved EBP
SavedEBP 0xcccccc
var1 0xcccccccc
buffer[128]
Shellcode: 0xcc...
%ebp
%esp
EAX
ESI
<strcpy>:
push %ebp
mov %esp, %ebp
mov 0x8(%ebp), %eax
mov 0xc(%ebp), %esi
. . .
pop %ebp
ret
[heap] rwxp

ret2strcpy


0xbfa7674c (src)
0x8ce40b0 (dest)
argv0 0x8ce40b0 (dest)
(&strcpy) Saved EBP
SavedEBP 0xcccccc
var1 0xcccccccc
buffer[128]
Shellcode: 0xcc...
%ebp
%esp
EAX
ESI
<strcpy>:
push %ebp
mov %esp, %ebp
mov 0x8(%ebp), %eax
mov 0xc(%ebp), %esi
. . . ; data copy instructions
pop %ebp
ret
strcpy
[heap] rwxp
Shellcode:
\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc
SavedEBP+dest+dest+src+morejunk

ret2strcpy

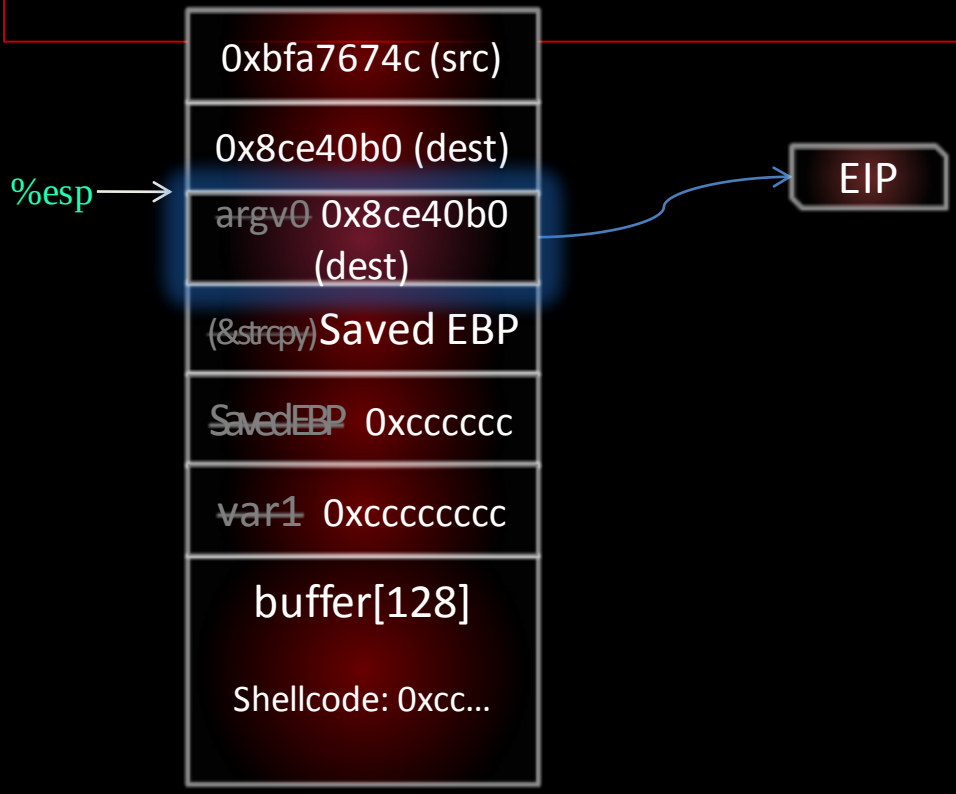



```
<strcpy>:
push    %ebp
mov     %esp, %ebp
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi

. . . ; data copy instructions

pop     %ebp
ret
```

[heap] rwxp

Shellcode:
\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc
SavedEBP+dest+dest+src+morejunk

ret2strcpy

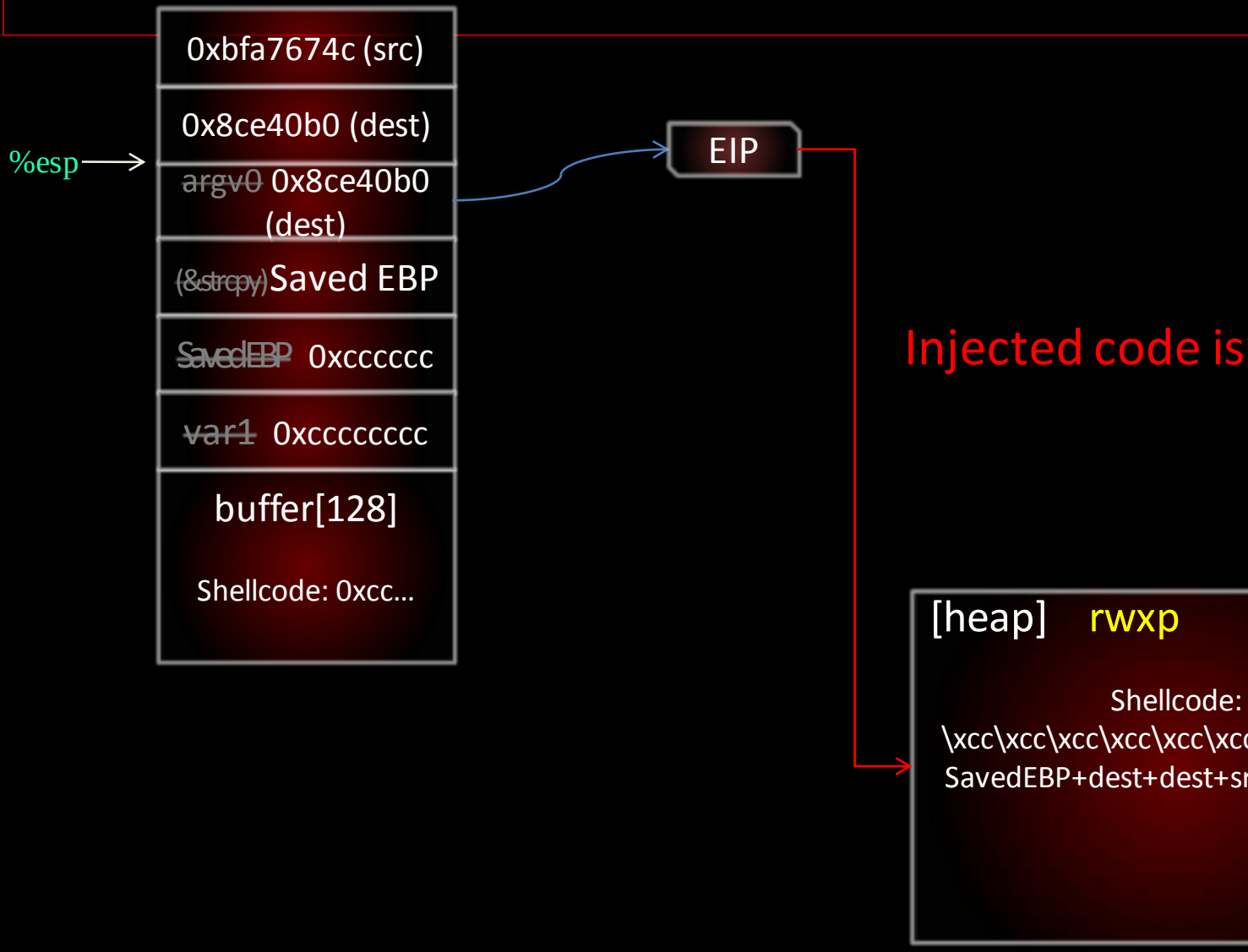



```
<strcpy>:
push    %ebp
mov     %esp, %ebp
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi

. . . ; data copy instructions

pop     %ebp
ret
```

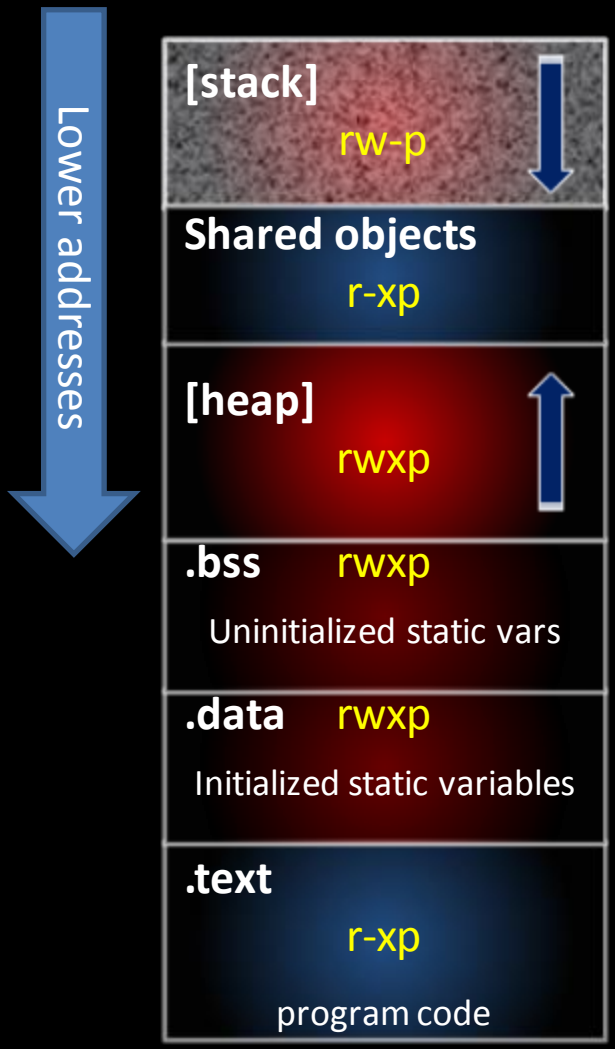
[heap] rwxp

Shellcode:
\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc
SavedEBP+dest+dest+src+morejunk

ret2strcpy


0xbfa7674c (src)
0x8ce40b0 (dest)
%esp
argv0 0x8ce40b0 (dest)
EIP
(&strcpy) Saved EBP
SavedEBP 0xcccccc
var1 0xcccccccc
buffer[128]
Shellcode: 0xcc...
Injected code is executed!
[heap] rwxp
Shellcode:
\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc
SavedEBP+dest+dest+src+morejunk

ret2strcpy

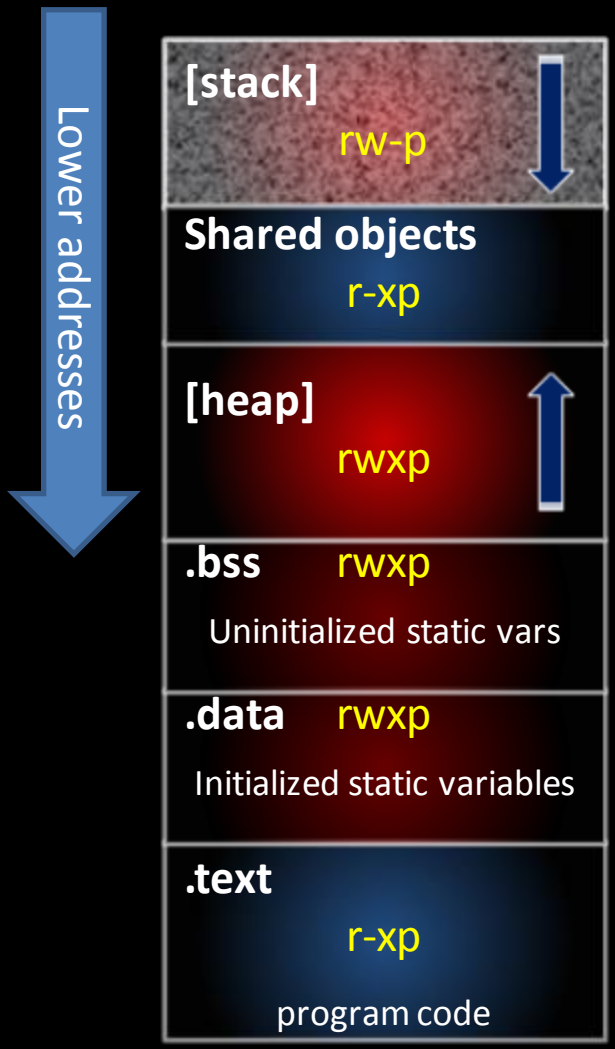


Shellcode is copied to any writable-and-executable memory region.

EIP is manipulated to redirect execution to shellcode in executable memory.

Mitigations???

ret2strcpy



Shellcode is copied to any writable-and-executable memory region.

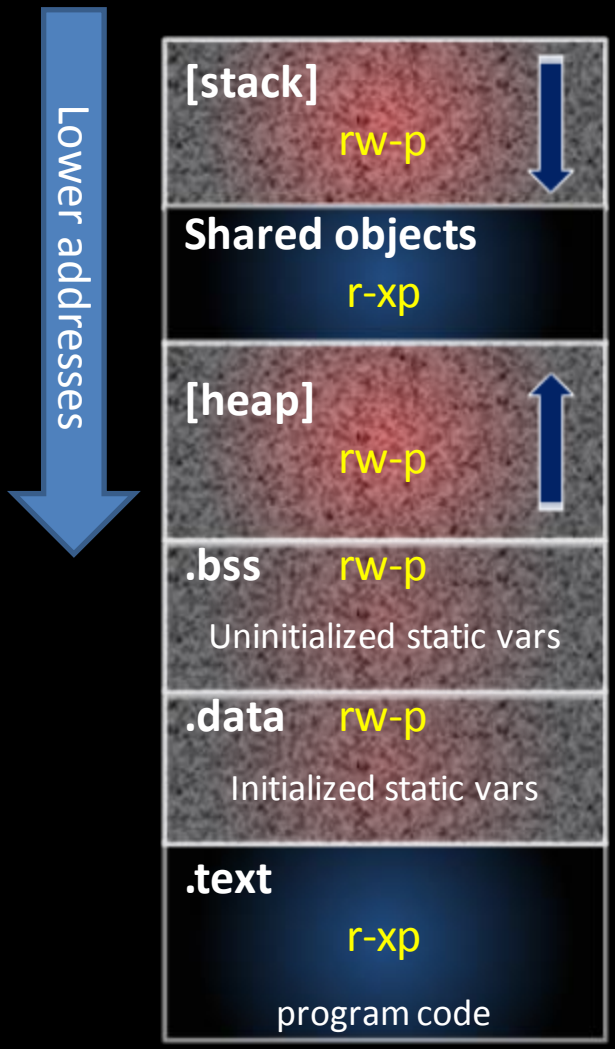
EIP is manipulated to redirect execution to shellcode in executable memory.

Mitigations???

Non-Executable Memory

Memory is either writable or executable, but not both

ret2strcpy



Shellcode is copied to any writable-and-executable memory region.

EIP is manipulated to redirect execution to shellcode in executable memory.

Mitigations???

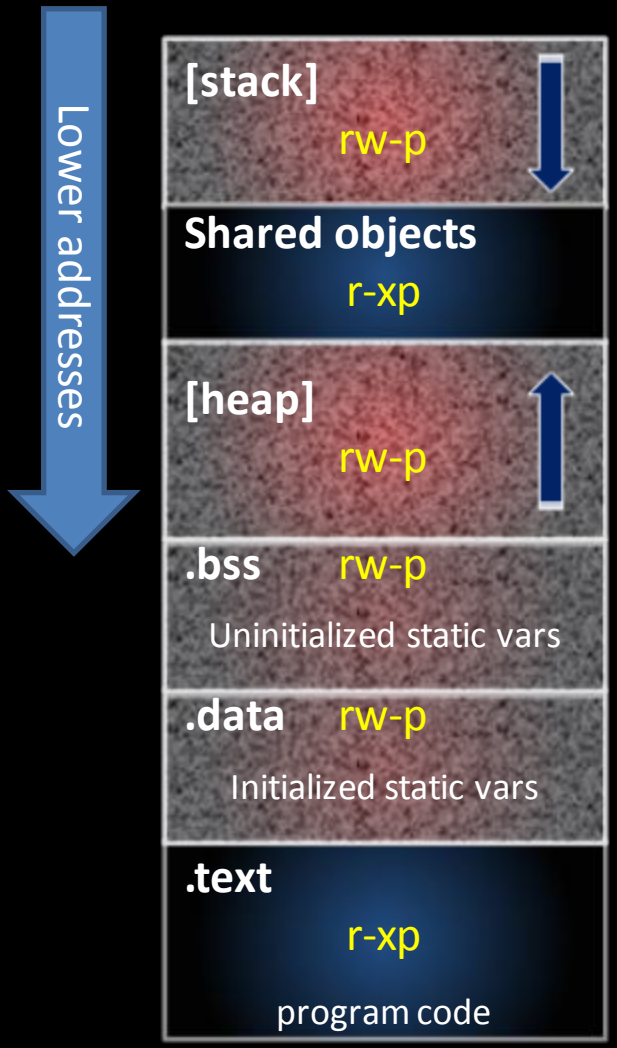
Non-Executable Memory

```
Memory is either writable
or executable, but not both
```

W^X

Shellcode can not be injected anymore

NX-Memory



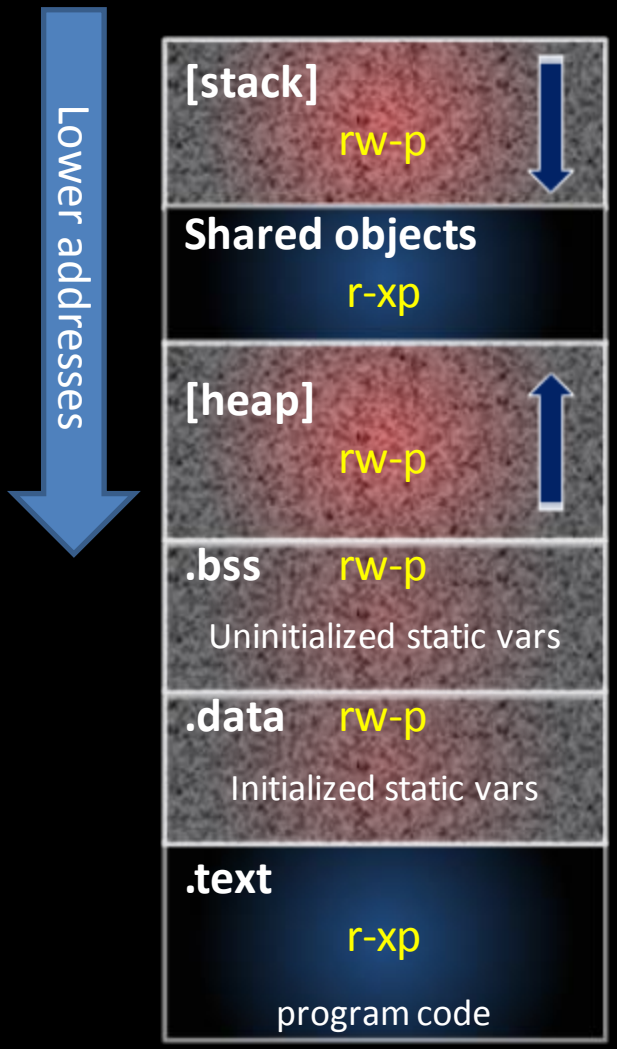
-> Linux: PaX, 2000
- Software emulation.

-> Windows: **Data Execution Prevention**
- Introduced in XP SP2
- Software and Hardware DEP

-> AMD: NX Bit
-> Intel: XD (eXecute Disable) Bit

Weakness?

NX-Memory



-> Linux: PaX, 2000
- Software emulation.

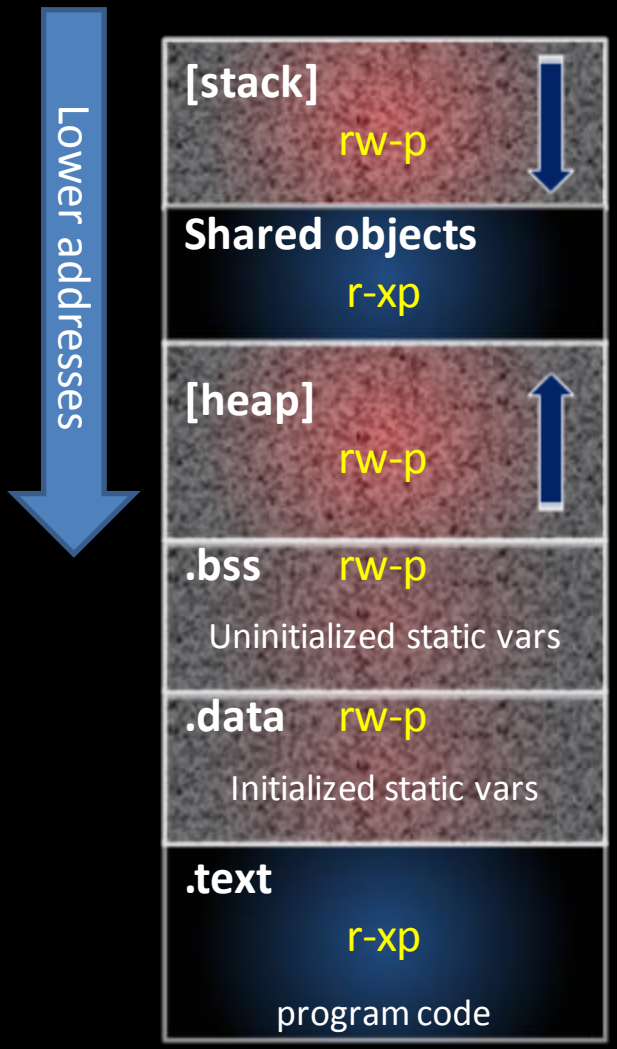
-> Windows: Data Execution Prevention
- XP SP2
- Software and Hardware DEP

-> AMD: NX Bit
-> Intel: XD (eXecute Disable) Bit

Weakness?

There is still «legitimate» and executable code in the binary and libraries that can be used.

ret2libc



libc – Standard C Library

Built-in standard functions which can be linked with any C program.

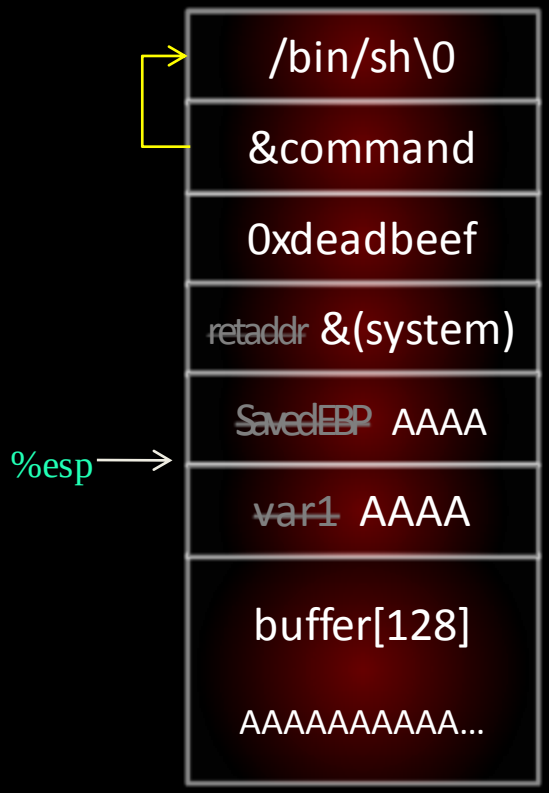
Return into a loaded library instead of injected code.

Frame faking:
Control data in the stack to set-up function arguments and return-addresses

i.e. `system()`

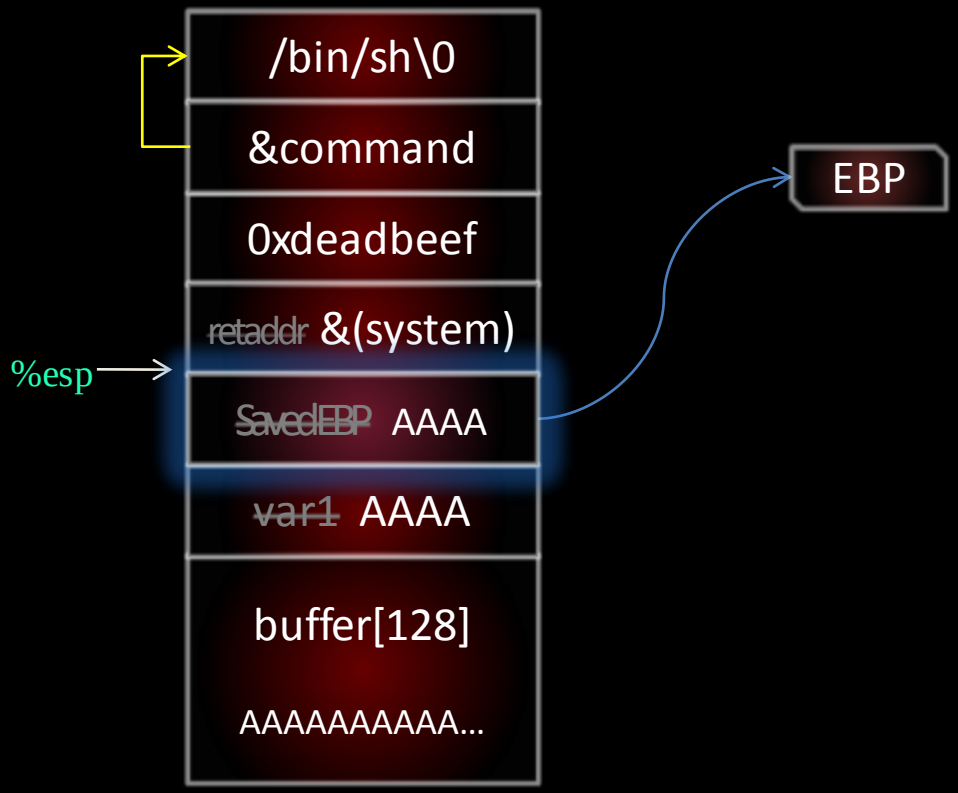
There is still «legitimate» and executable code in the binary and libraries that can be used.

ret2libc



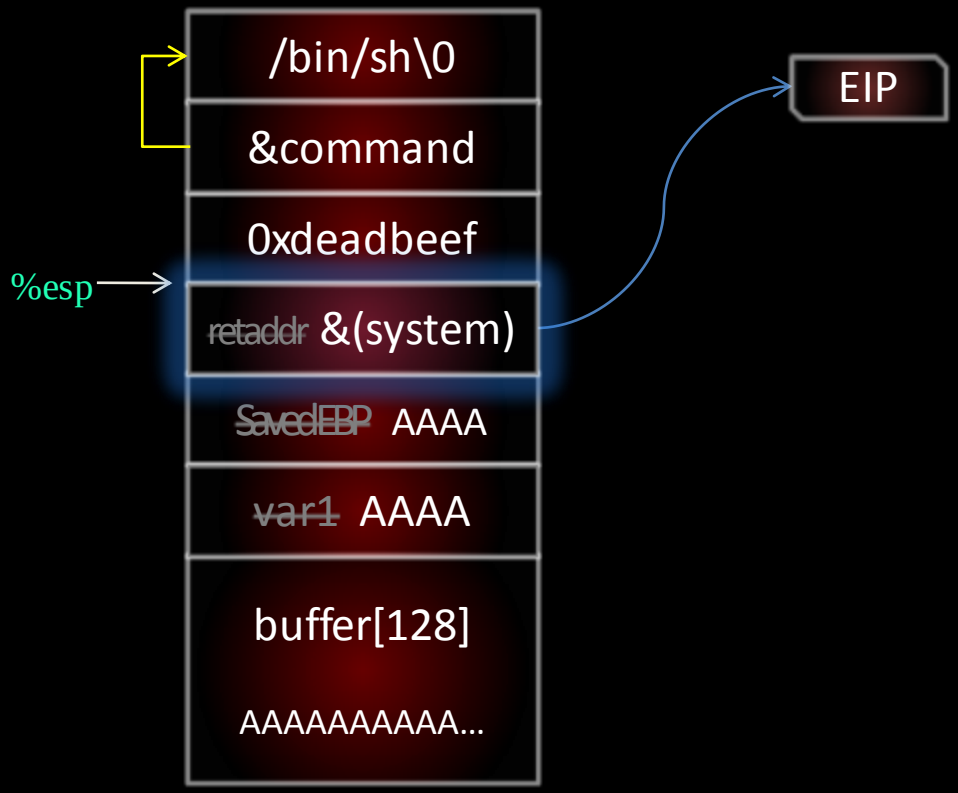

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

ret2libc



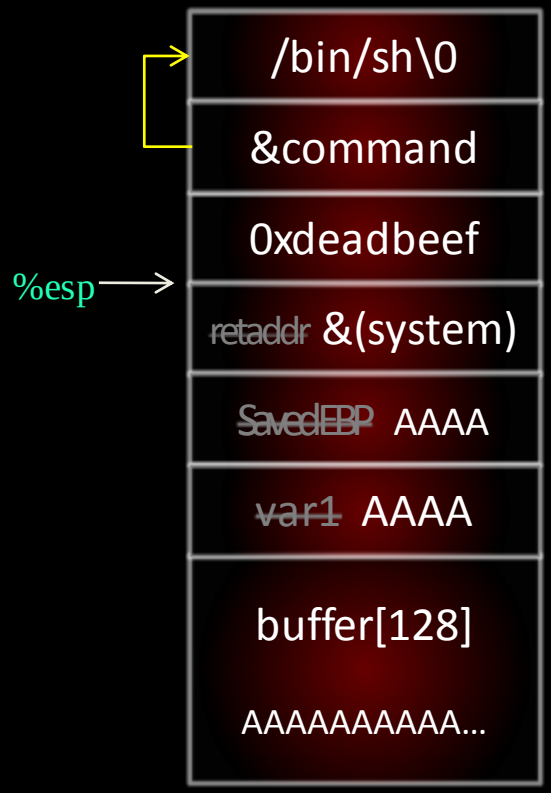

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

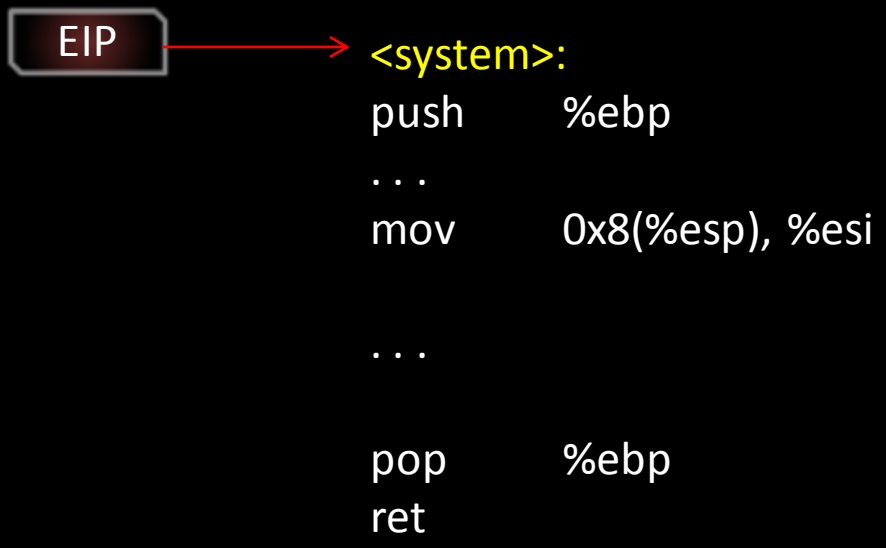
ret2libc



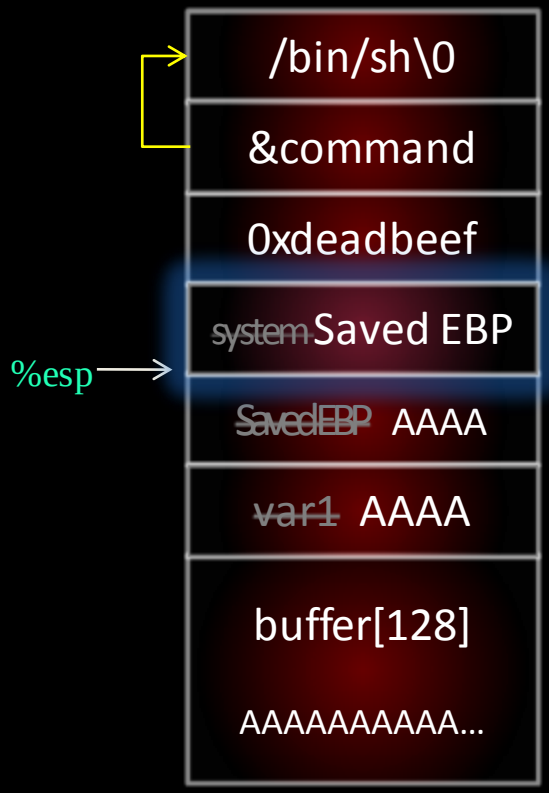

```
int  vulnerable(char
*argv0)
{
      int var1;
      char buffer[128];

      strcpy (buffer, argv0);
      return 0;
}

      movl   %ebp, %esp
      pop    %ebp
      ret
```

ret2libc


/bin/sh\0
&command
0xdeadbeef
%esp
retaddr &(system)
SavedEBP AAAA
var1 AAAA
buffer[128]
AAAAAAAAAA...

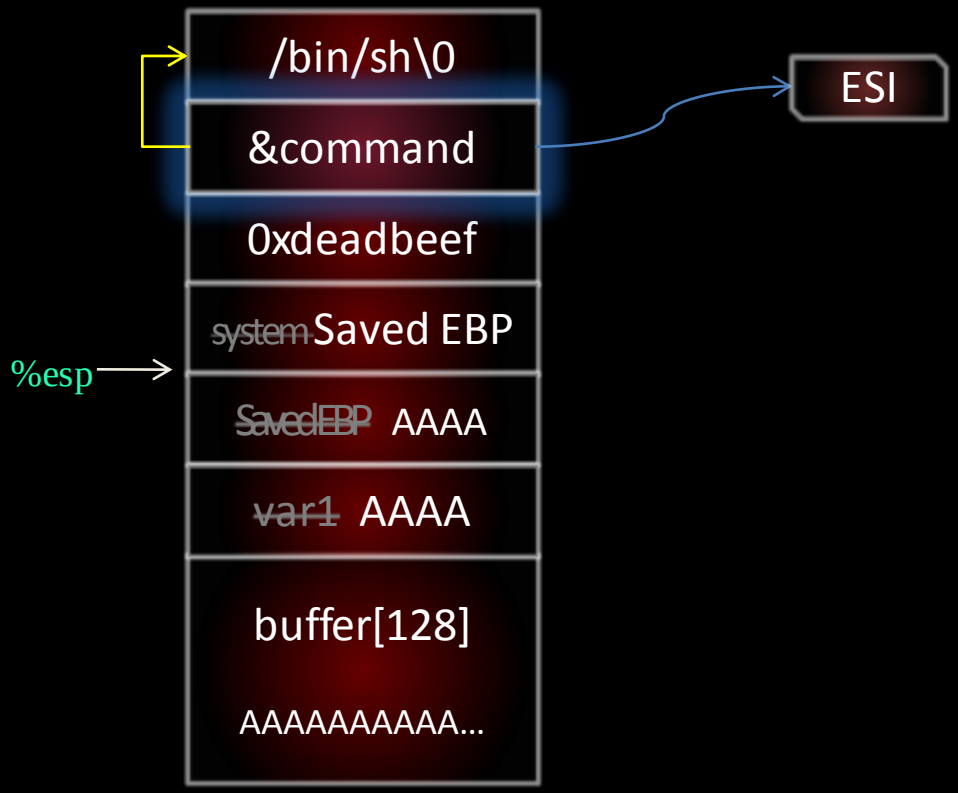

EIP
<system>:
push %ebp
. . .
mov 0x8(%esp), %esi
. . .
pop %ebp
ret

ret2libc



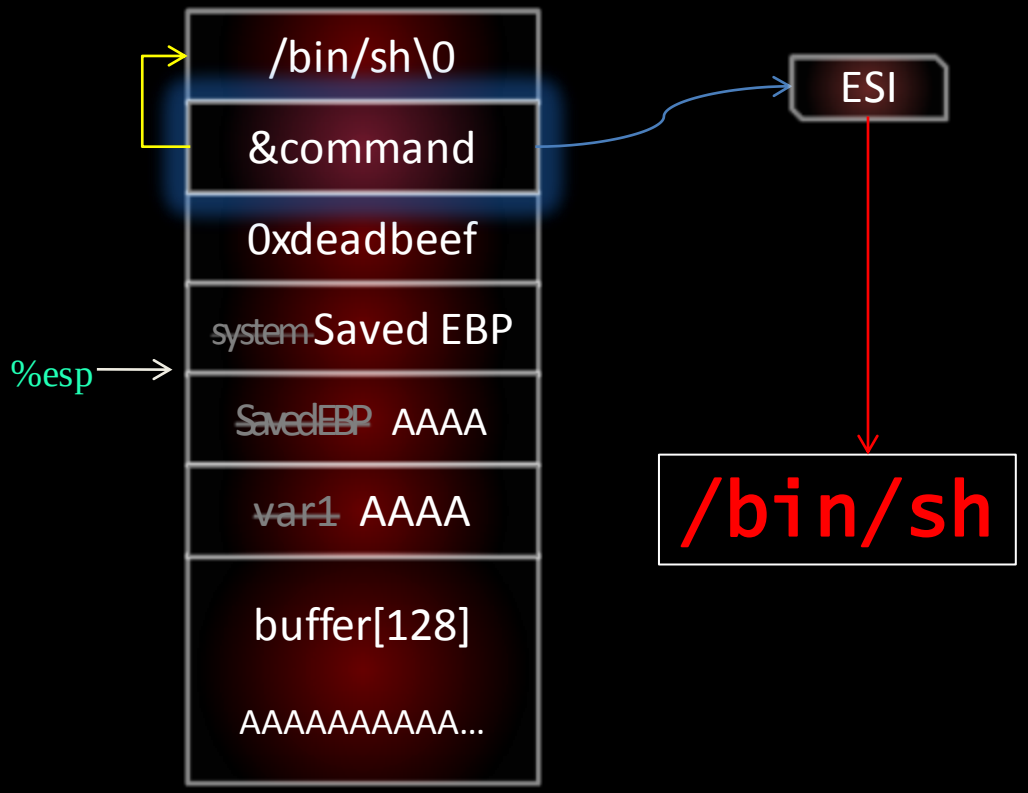

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . .

pop     %ebp
ret
```

ret2libc




```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . .

pop     %ebp
ret
```

ret2libc




```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

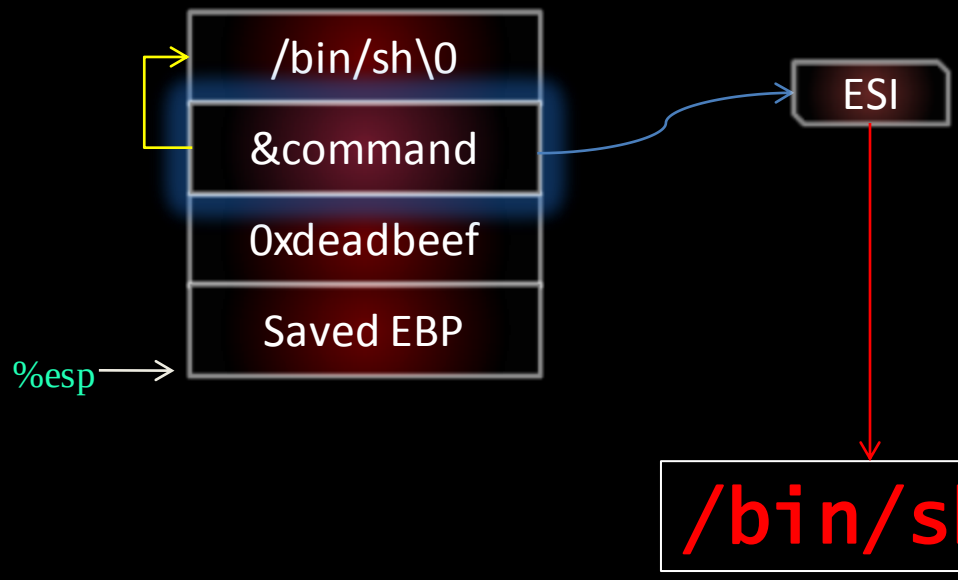
ret2libc

Limitations:

Only one function can be called

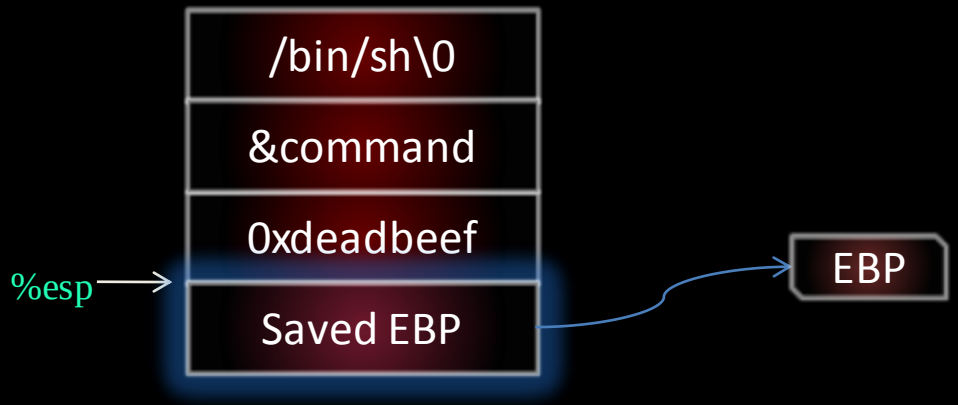
Usually more than one function is needed for complex computations.

Return Chaining



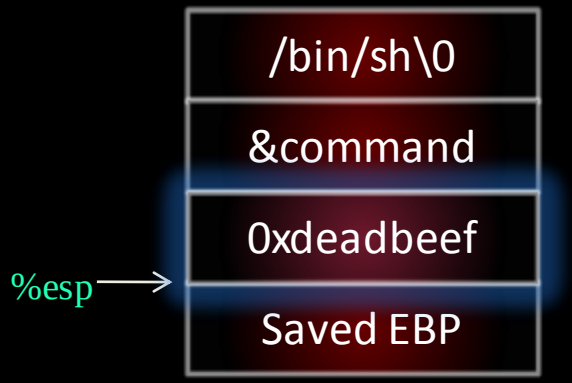

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

Return Chaining



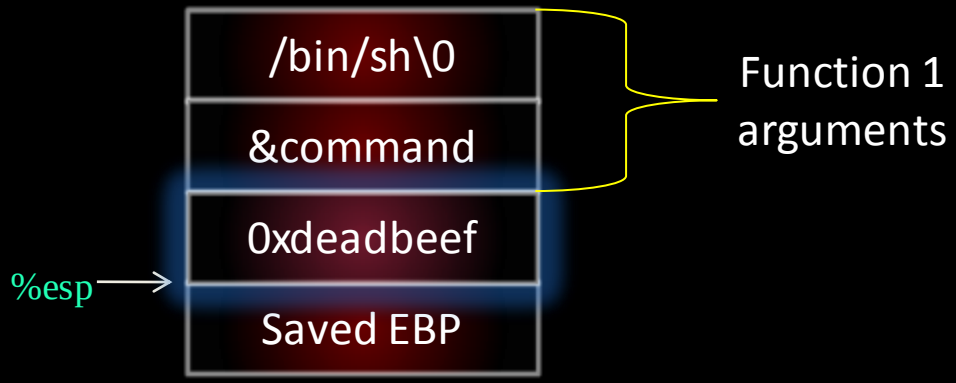

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

Return Chaining



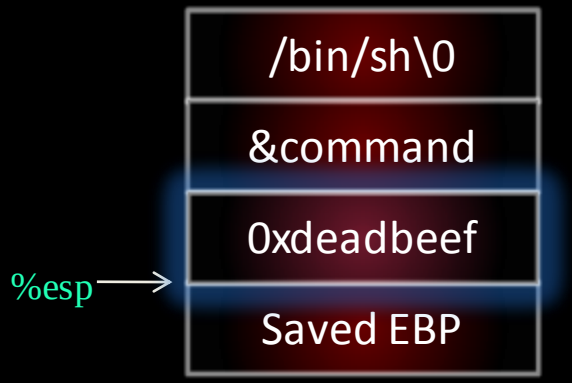

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

Return Chaining



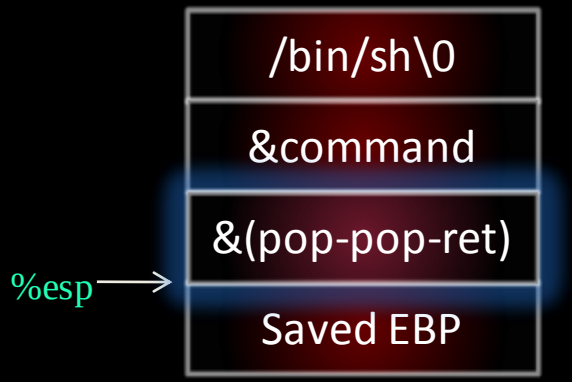

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

Return Chaining



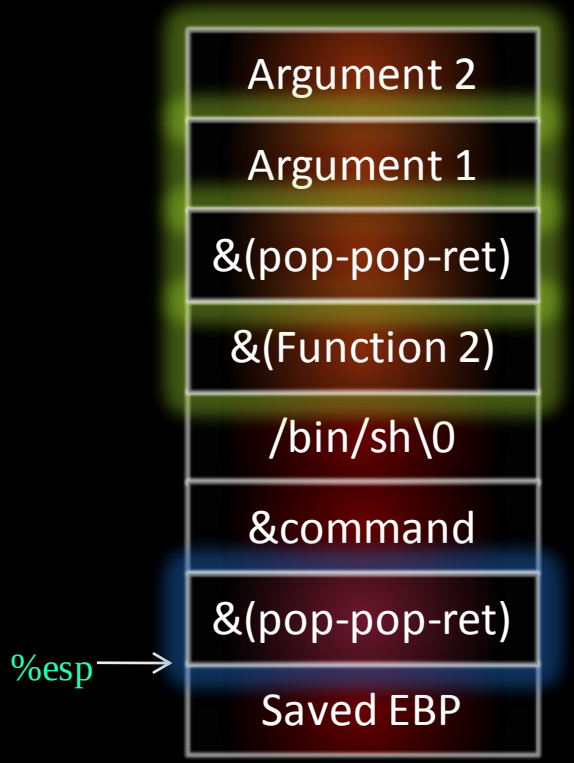

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

Return Chaining



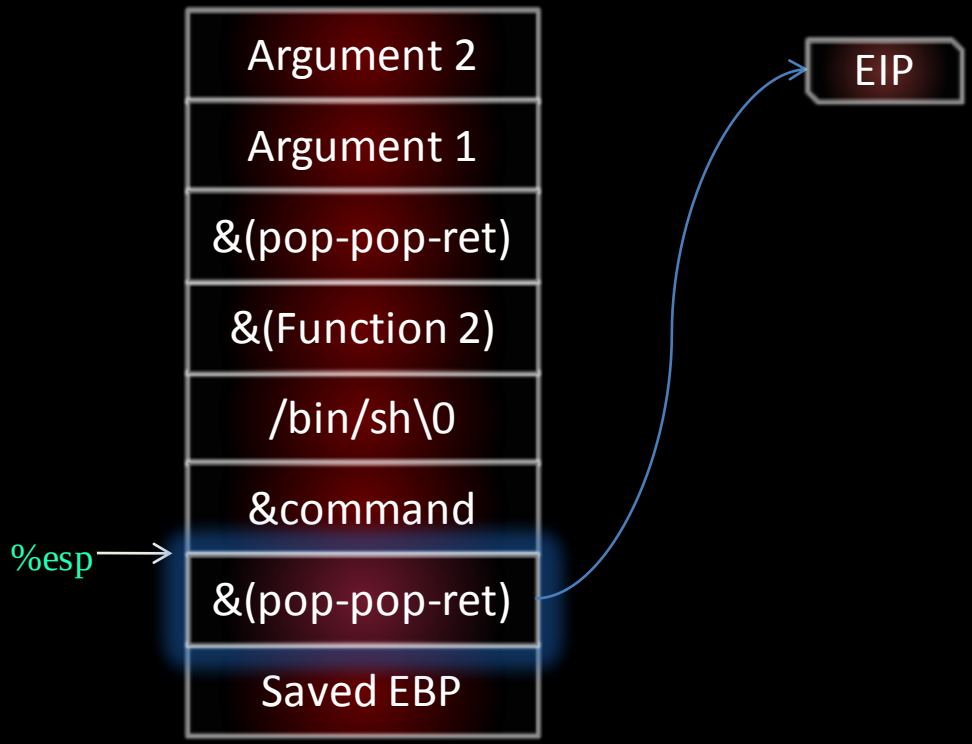

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

Return Chaining



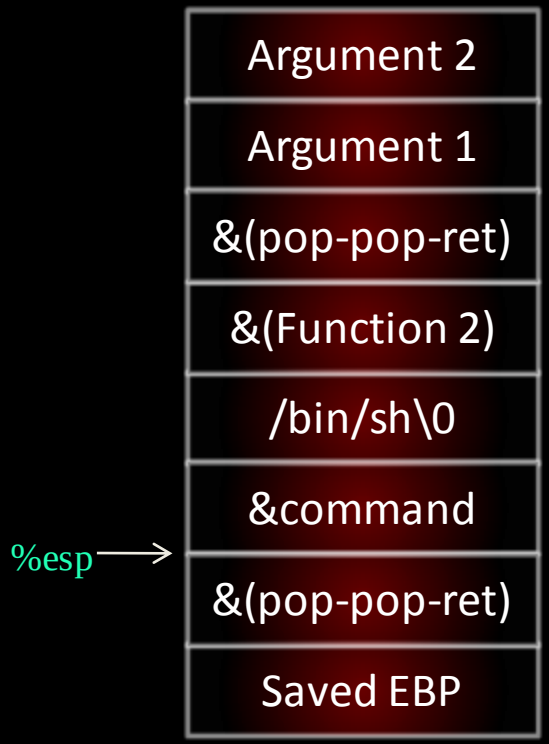

```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

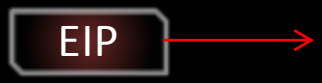
Return Chaining




```
<system>:
push    %ebp
. . .
mov     0x8(%esp), %esi

. . . ; Execute command

pop     %ebp
ret
```

Return Chaining

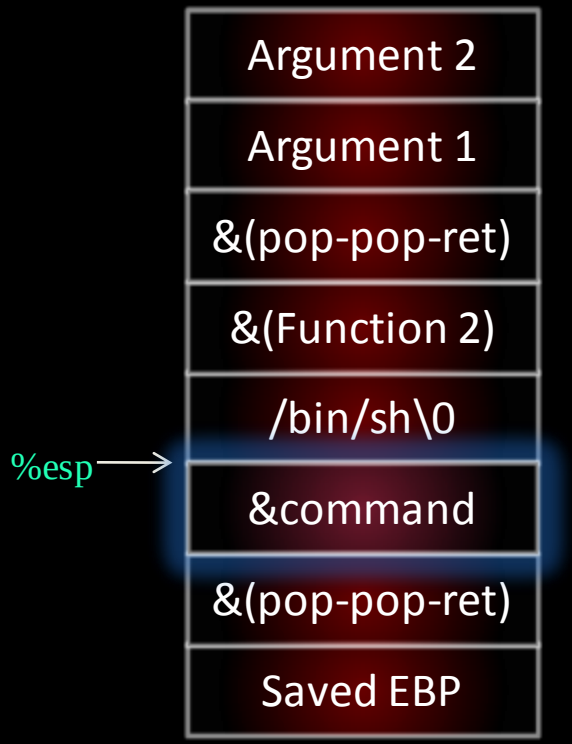




<0x0805162c>:

```
pop    %esi
pop    %ebx
ret
```

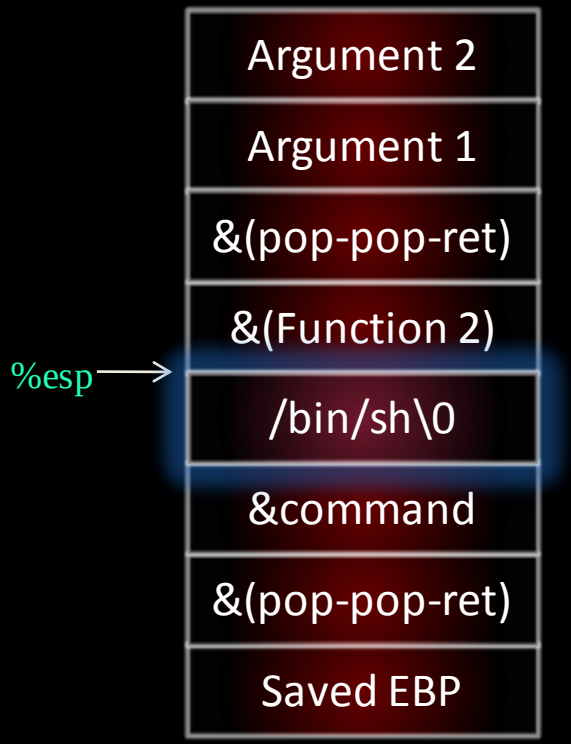
Return Chaining



<0x0805162c>:

```
pop    %esi
pop    %ebx
ret
```

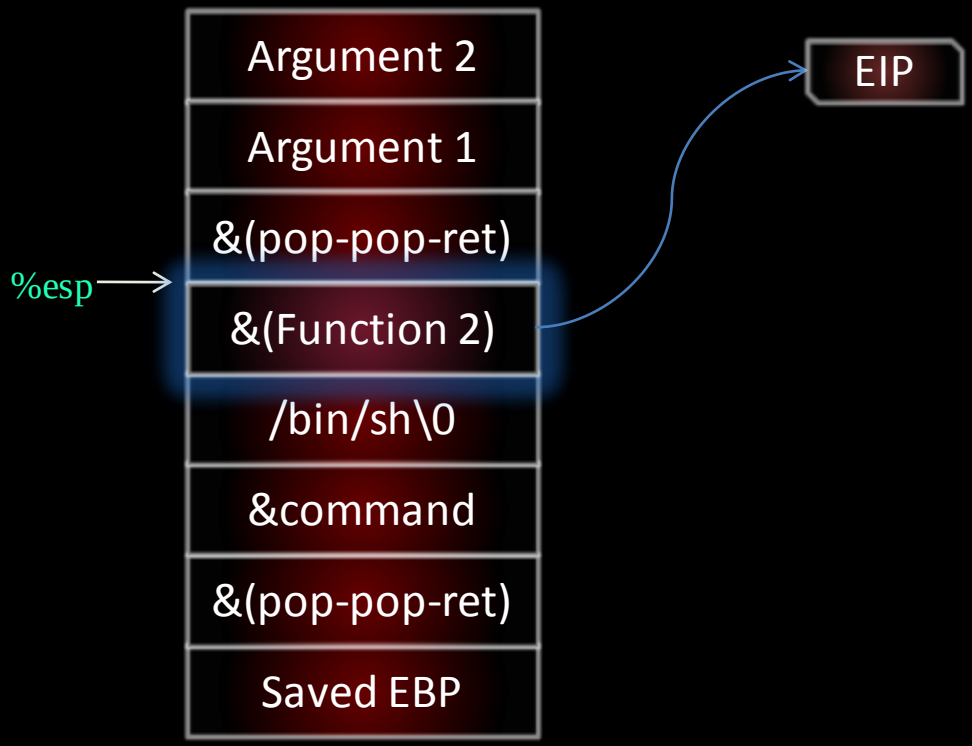
Return Chaining



<0x0805162c>:

```
pop    %esi
pop    %ebx
ret
```

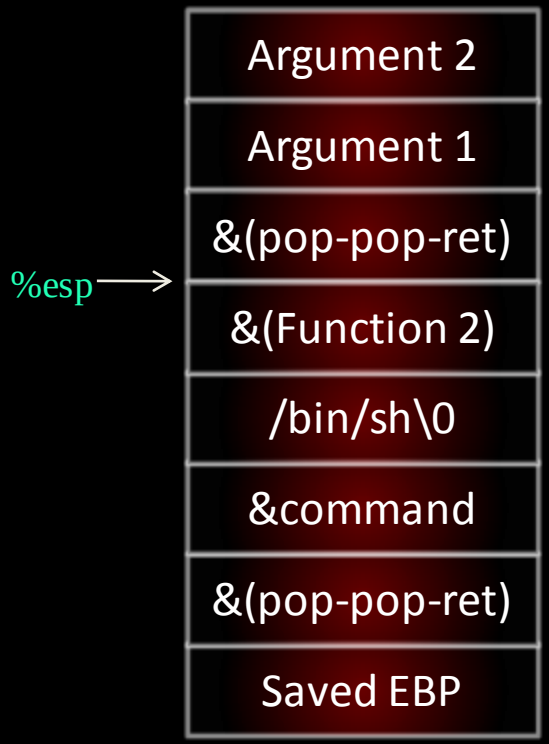
Return Chaining



<0x0805162c>:

```
pop    %esi
pop    %ebx
ret
```

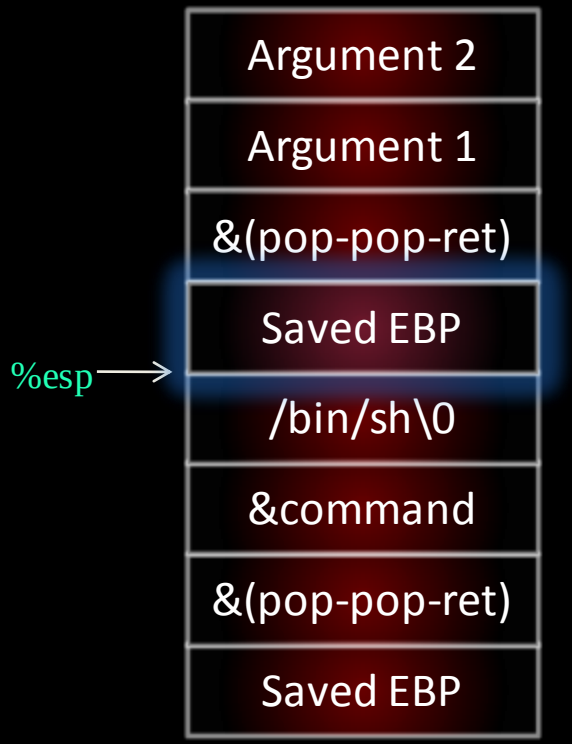
Return Chaining



<Function 2>:

```
push   %ebp
mov    %esp, %ebp
mov    0x8(%ebp), %eax
mov    0xc(%ebp), %esi
. . .
pop    %ebp
ret
```

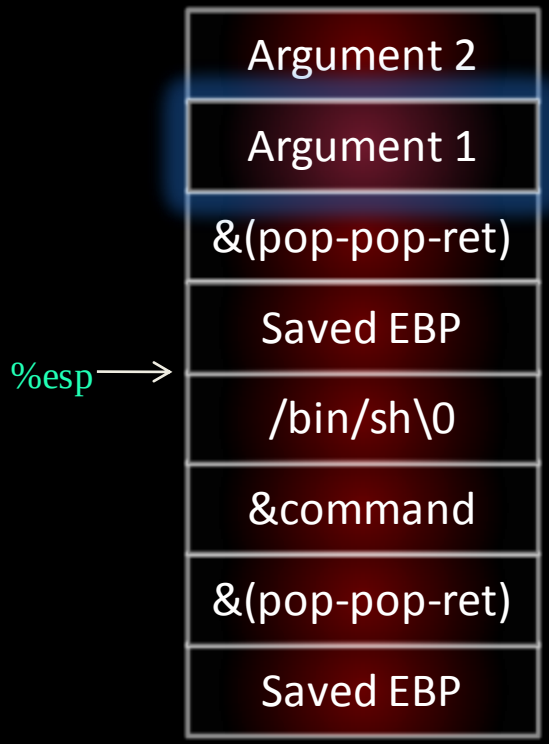
Return Chaining



<Function 2>:

```
push   %ebp
mov    %esp, %ebp
mov    0x8(%ebp), %eax
mov    0xc(%ebp), %esi
. . .
pop    %ebp
ret
```

Return Chaining



<Function 2>:

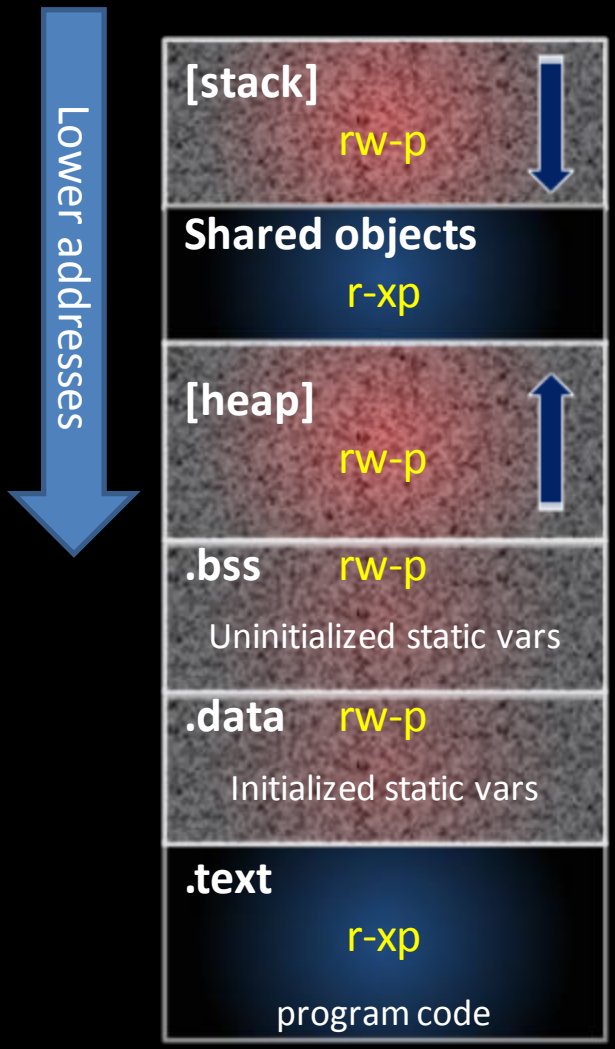
```
push    %ebp
mov     %esp, %ebp
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . .
pop     %ebp
ret
```

Return Chaining

<Function 2>:

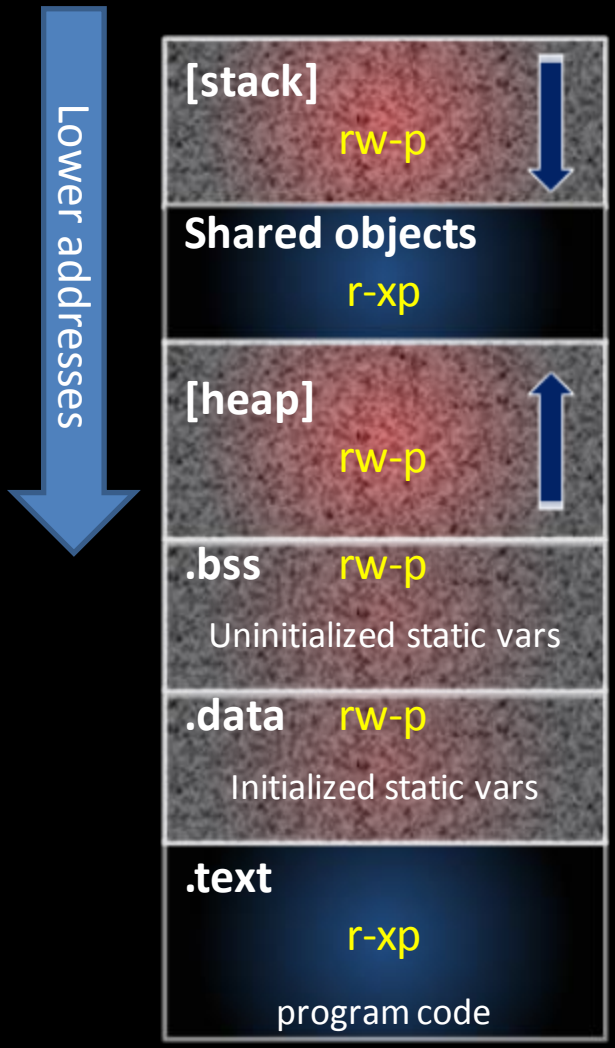
```
push    %ebp
mov     %esp, %ebp
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . .
pop     %ebp
ret
```

NX Memory



We can defeat NX-Memory by jumping to code located in the **binary** and **shared libraries.**

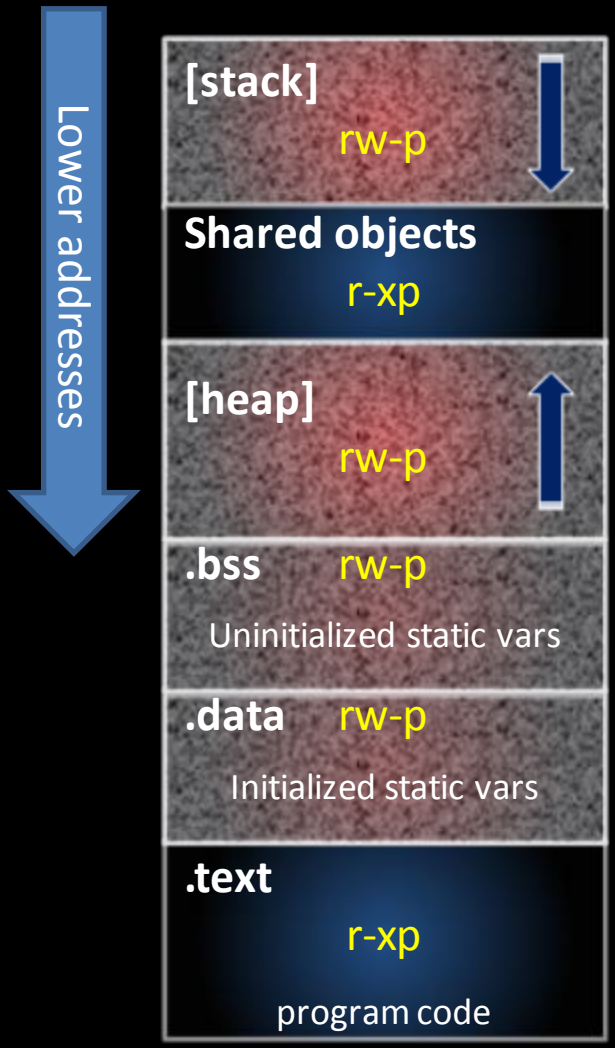
NX Memory



We can defeat NX-Memory by jumping to code located in the **binary** and **shared libraries.**

- **ret2libc**
- **Return Chaining**
- **ret2text**
- **ret2plt**
- **ret2dl-resolve**

NX Memory

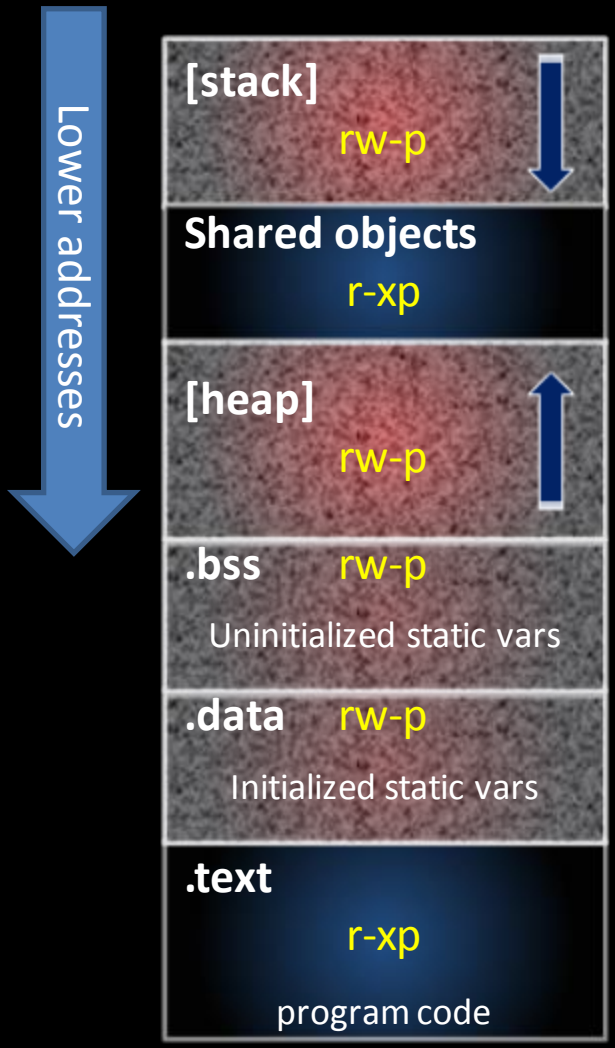


We can defeat NX-Memory by jumping to code located in the **binary** and **shared libraries.**

- **ret2libc**
- **Return Chaining**
- **ret2text**
- **ret2plt**
- **ret2dl-resolve**

Downsides?

NX Memory



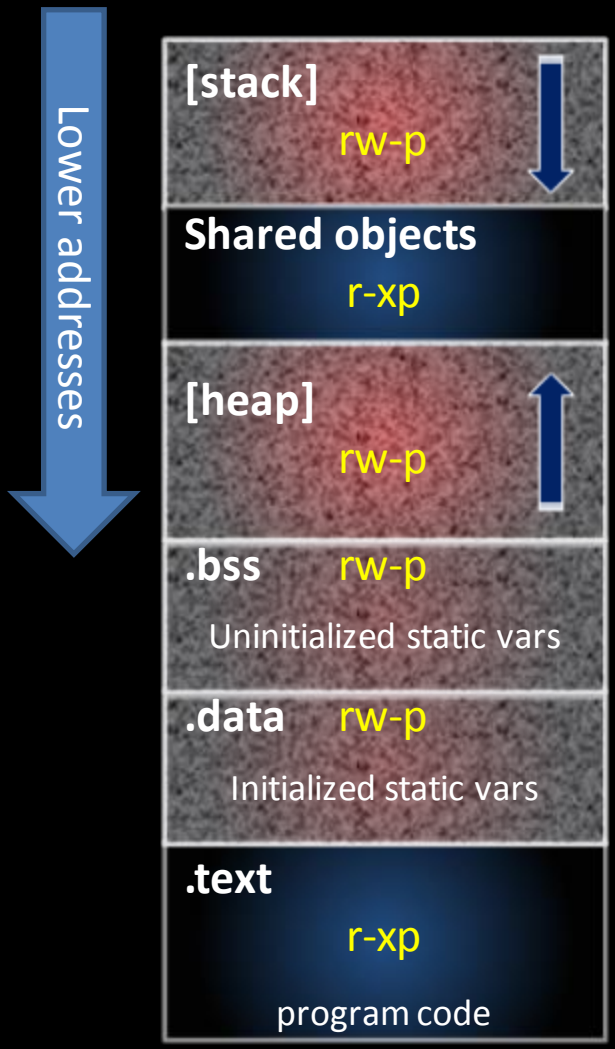
We can defeat NX-Memory by jumping to code located in the **binary** and **shared libraries.**

- **ret2libc**
- **Return Chaining**
- **ret2text**
- **ret2plt**
- **ret2dl-resolve**

Downsides?

- It's rare to find useful functions in the `.text` section
- Dependency on code provided by libraries
  - Remove dangerous instructions? i.e. `system()`
- It is always nice to have injected code (a.k.a shellcode)
- Complex operations are hard through ret-chaining
- Forget about having custom code

NX Memory



We can defeat NX-Memory by jumping to code located in the **binary** and **shared libraries.**

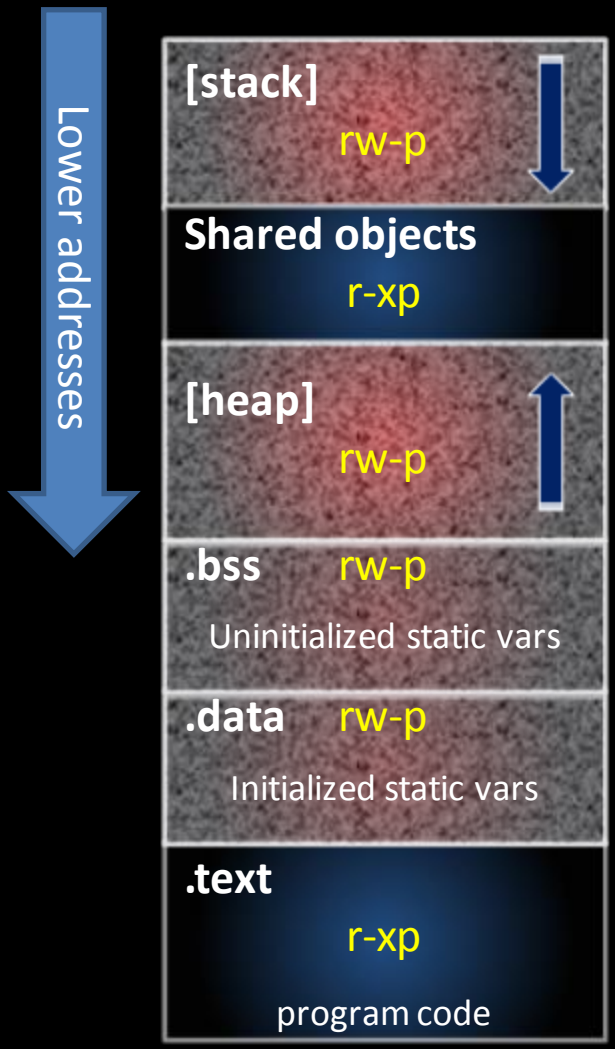
- **ret2libc**
- **Return Chaining**
- **ret2text**
- **ret2plt**
- **ret2dl-resolve**

Downsides?

- It's rare to find useful functions in the `.text` section
- Dependency on code provided by libraries
  - Remove dangerous instructions? i.e. `system()`
- It is always nice to have injected code (a.k.a shellcode)
- Complex operations are hard through ret-chaining
- Forget about having custom code

RISK

NX Memory

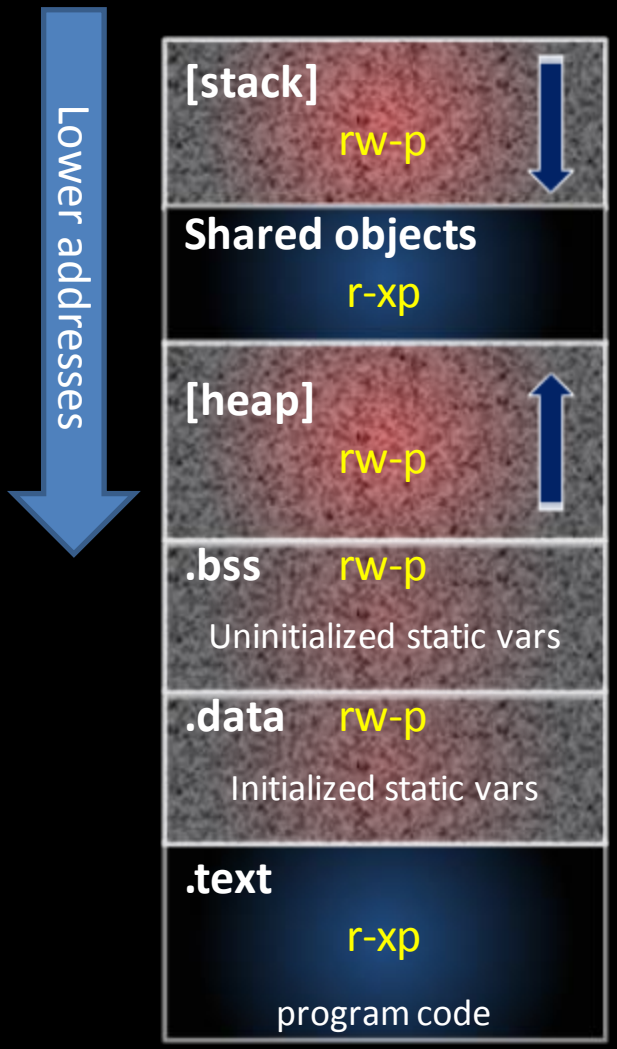


We can defeat NX-Memory by jumping to code located in the **binary** and **shared libraries.**

- **ret2libc**
- **Return Chaining**
- **ret2text**
- **ret2plt**
- **ret2dl-resolve**

Mitigations???

NX Memory



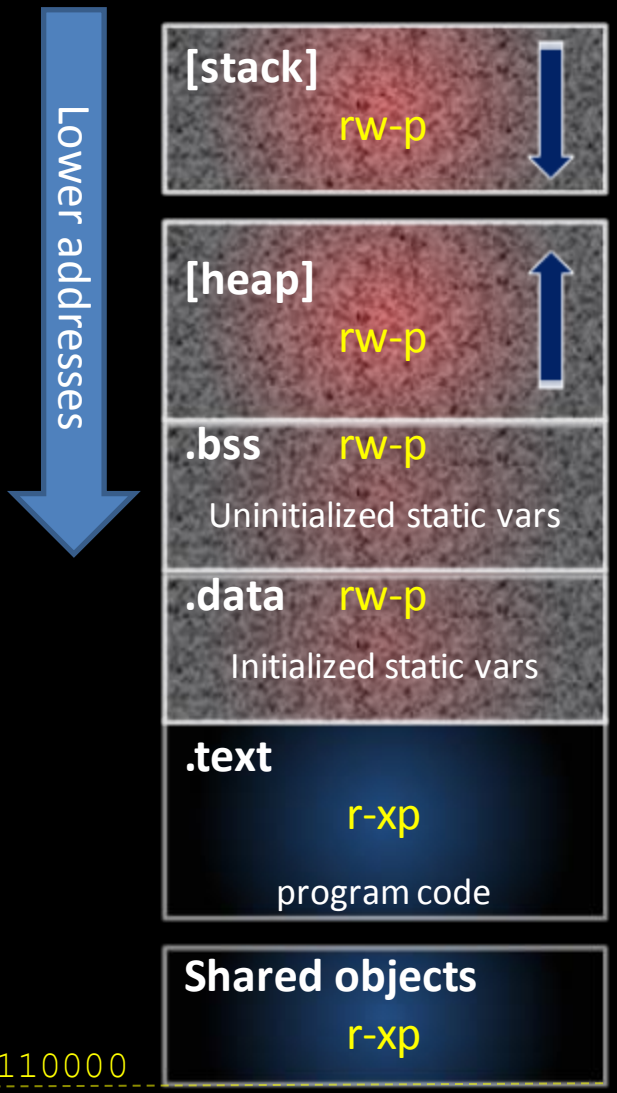
Mitigations???

AAAS

ASCII Armored Address Space

Linux Kernel Patch by Solar Designer

NX Memory



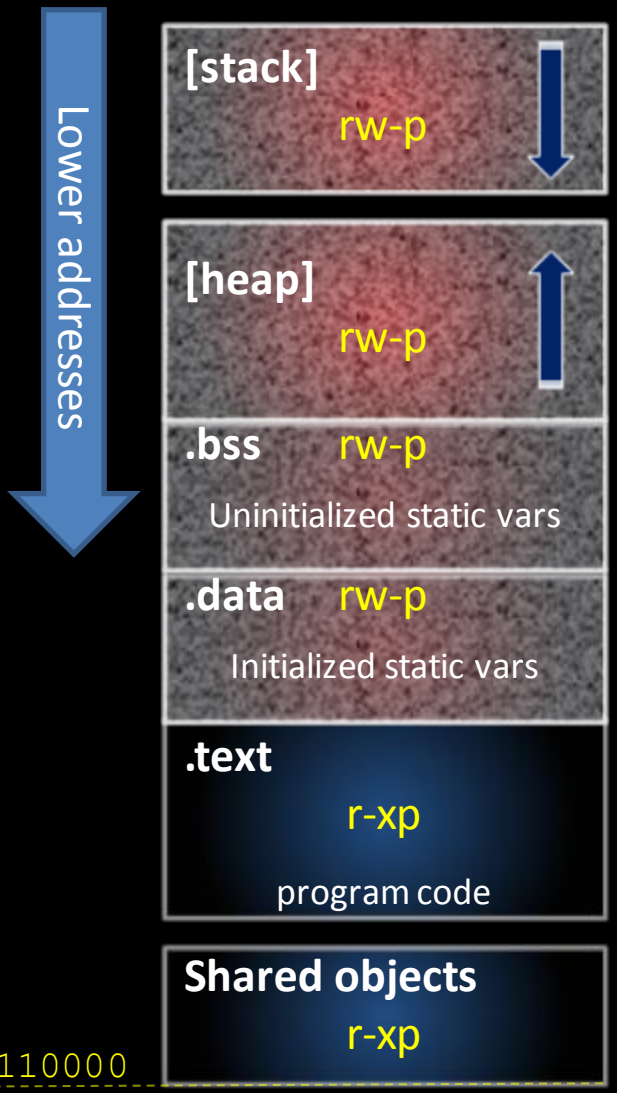
Mitigations???

AAAS

ASCII Armored Address Space

Linux Kernel Patch by Solar Designer

NX Memory



Mitigations???

AAAS

ASCII Armored Address Space

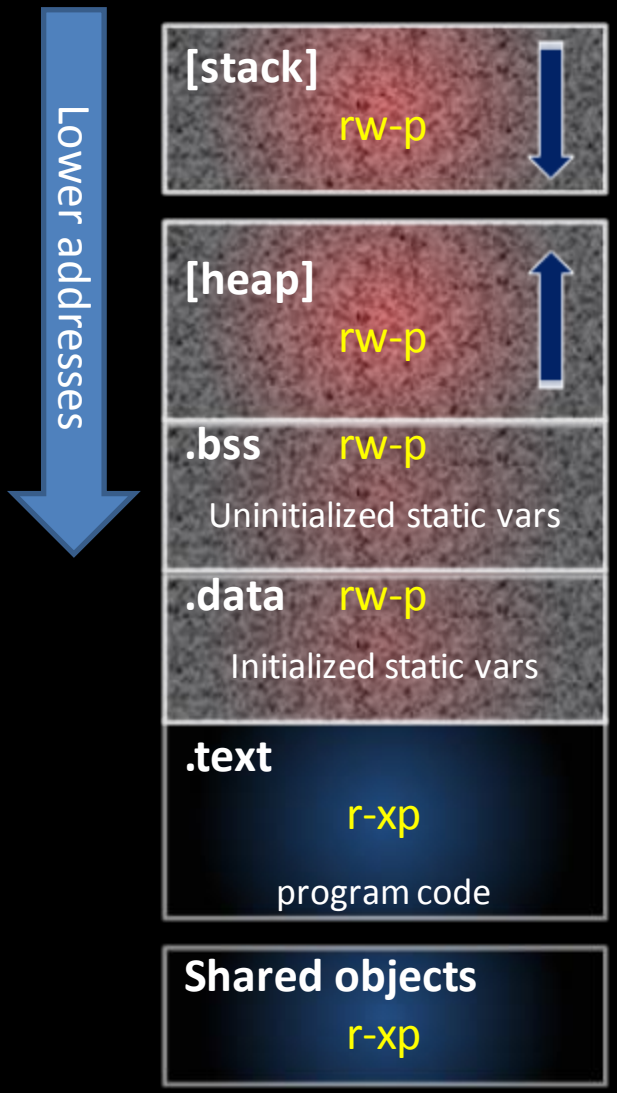
Linux Kernel Patch by Solar Designer

Loads shared libraries in memory addresses starting with NULL byte (‘\x00’)

Functions such as gets() and strcpy() stop operating when NULL byte is reached.

Still bypassable with **partial overwritting**.

NX Memory



Mitigations???

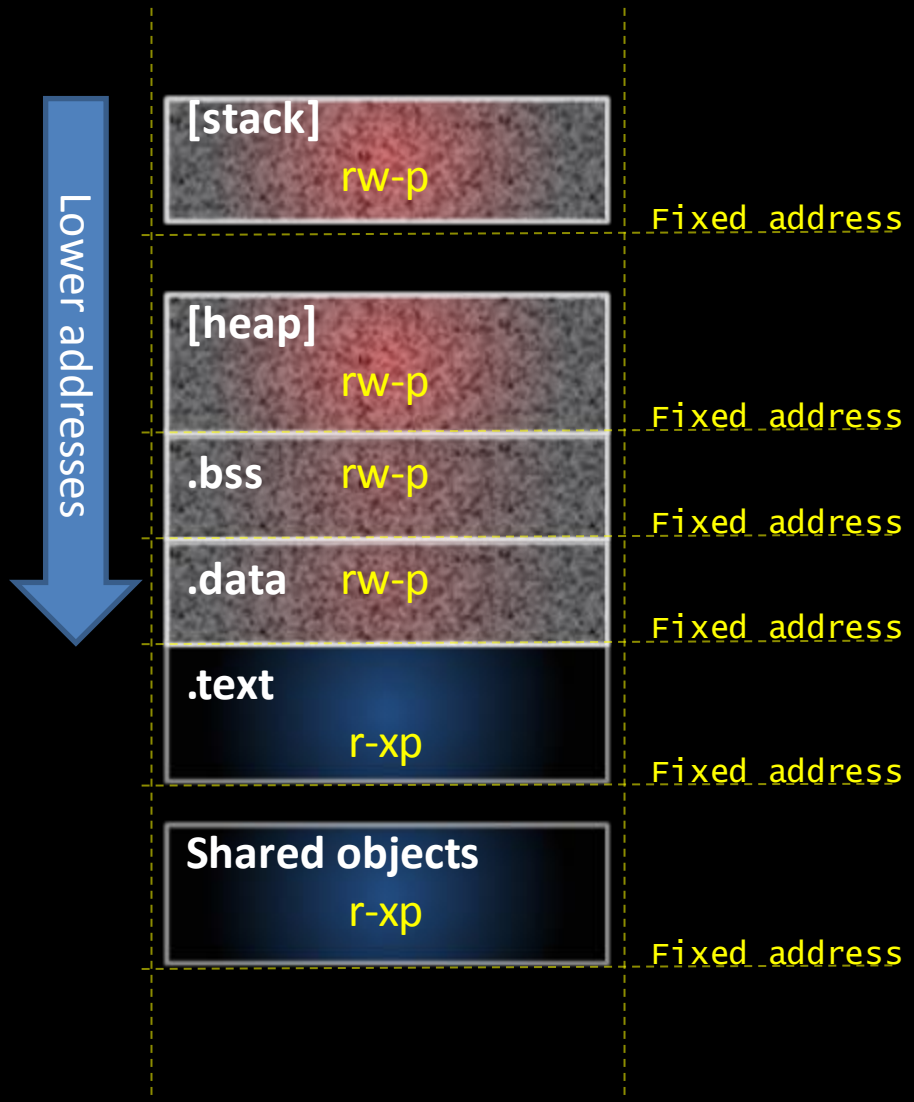
The feared…

and hated…

ASLR

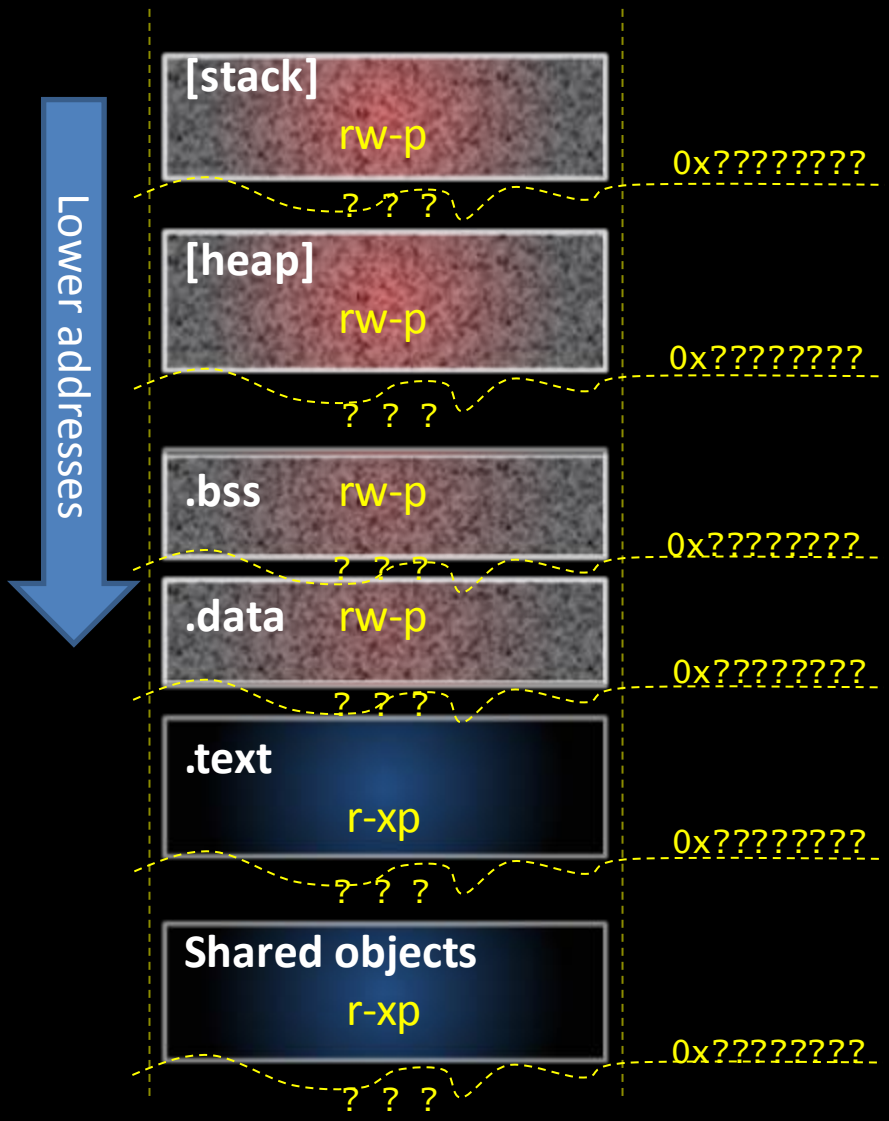
Address Space Layout Randomization

Address-Space Layout Randomization



Unless every address is unknown and unpredictable, there's always going to be room for some kind of attack.

Address-Space Layout Randomization



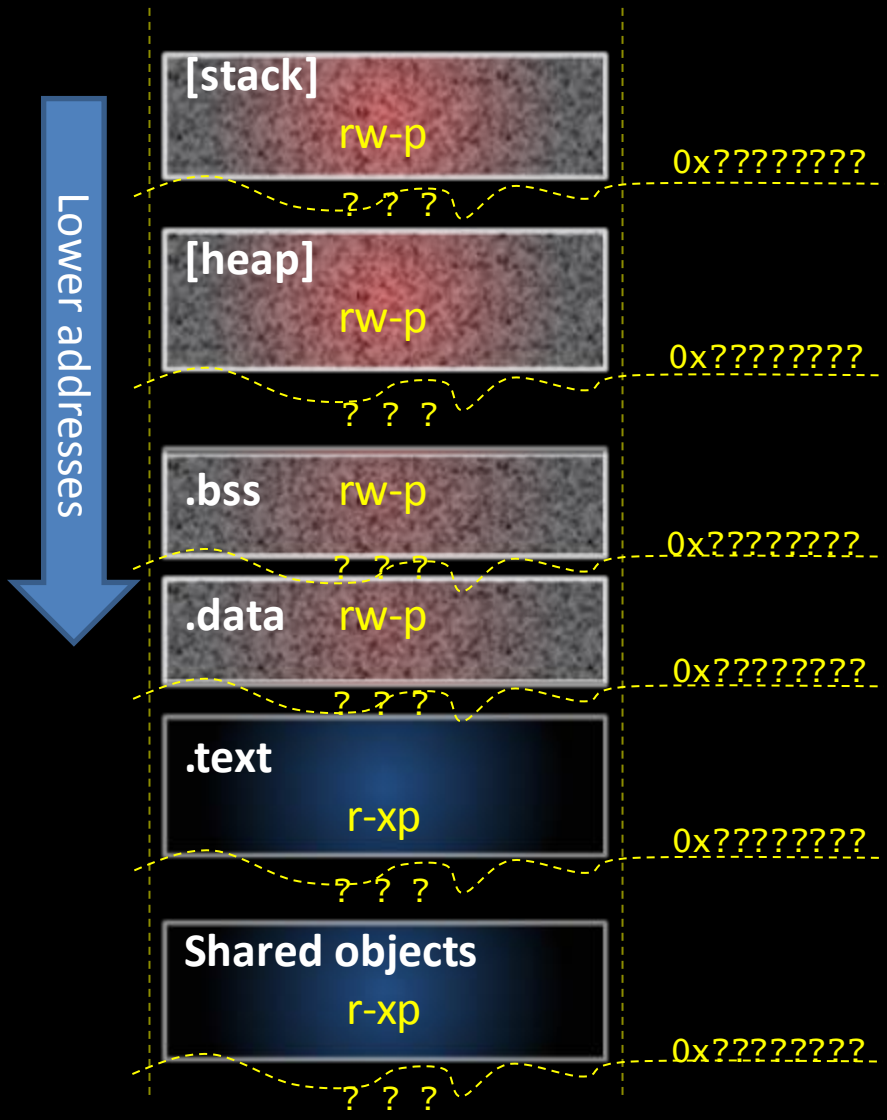
If the address of everything is randomized:

-> We can no longer jump reliably into shellcode in a stack/heap/PEB

-> We can no longer jump reliably into an instruction in a shared library.

-> We can no longer know where to point our payload's pointers.

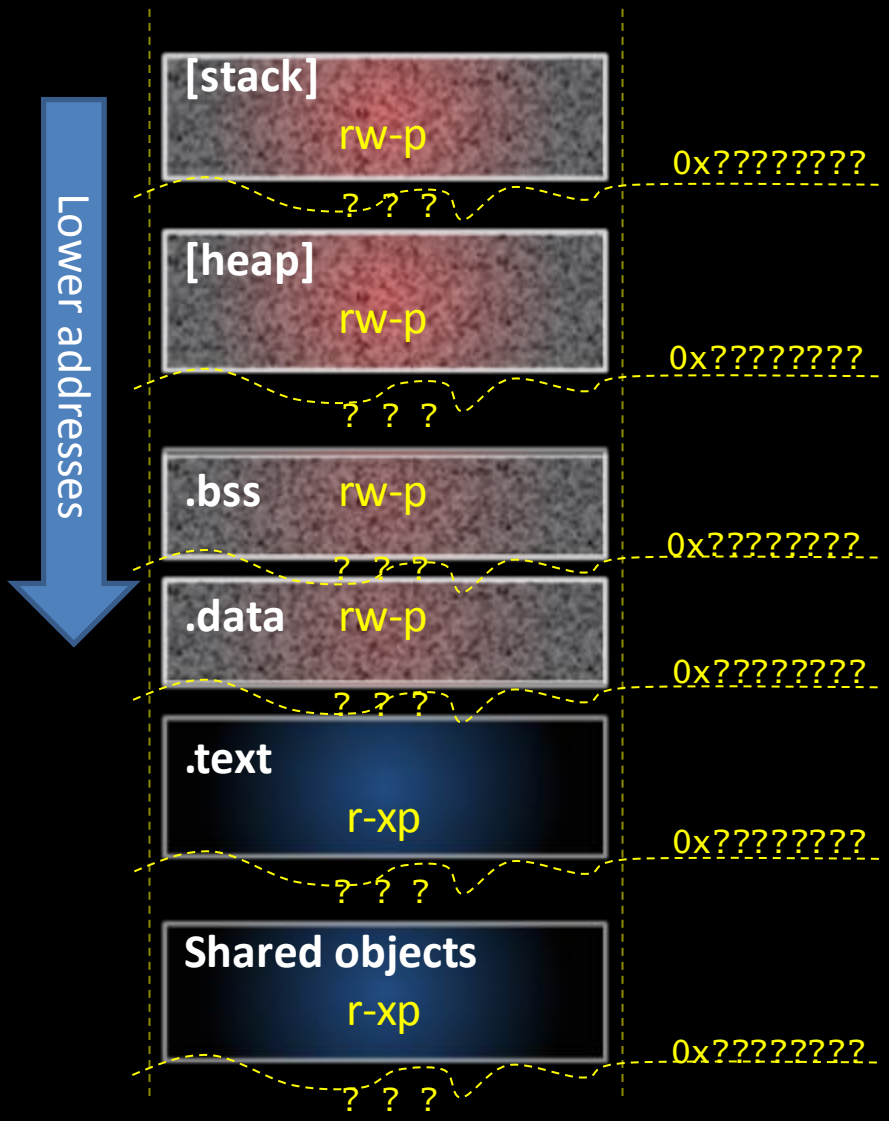
Address-Space Layout Randomization



Possible circumventions:

- Brute forcing
- Reducing entropy
- Memory layout disclosure bugs

Address-Space Layout Randomization



Introduced to **Linux** by PaX in 2001

Implemented in **Windows** by default since Vista (2007).

Weak implementation in **Mac OS X**

Return Oriented Programming

The Shellcode Necromancer

Why ROP?

- Return-into-libc (and similar) attacks are considered much more limited than code injection.

- NX Memory protections seem to be weakening attackers.

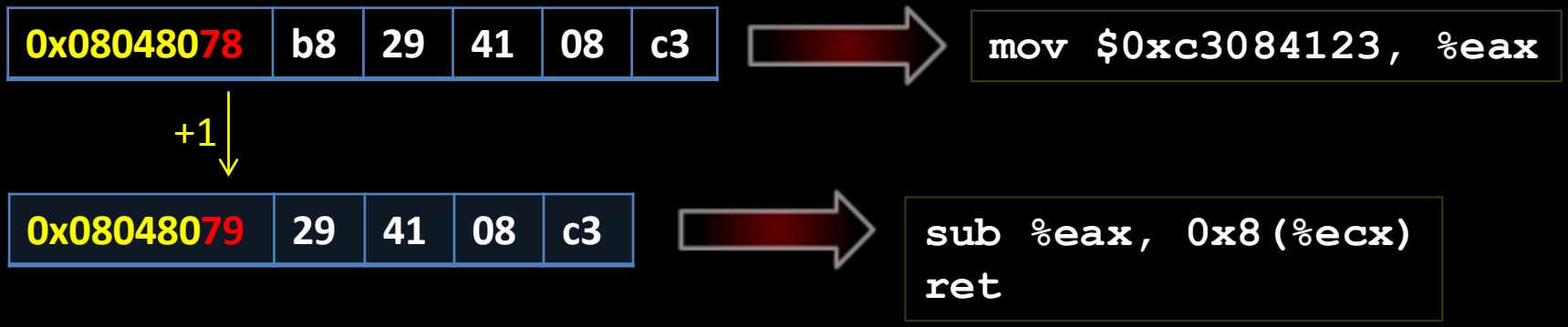
Work on ROP

- Based on the ideas of Sebastian Krahmer's 'borrowed code chunks' exploitation technique
- Original idea: *´The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*' – Hovav Shacham
- Smart ROP (Zynamics) `*Everybody be cool, this is a roppery*!' – Iozzo, Kornau, Weinmann
- Work in Linux '*Payload Already Inside: Data Re-use for ROP exploits*' – Long Le

What is ROP?

- A technique that turns return-into-code attacks as powerful as code injection.
  - Defeats NX Memory
  - Independent of code provided by libs and bins
  - Calls no functions at all
  - Uses instructions not placed by the assembler
  - Allows custom execution (shellcode necromancy)

Use of unintended instructions



- It may not be clear when does an instruction start or stop
- Finding unintended instructions by returning to the bytes in the middle of an instruction
- Takes advantage of X86 ISA ambiguity

Execution Flow

It is assumed that

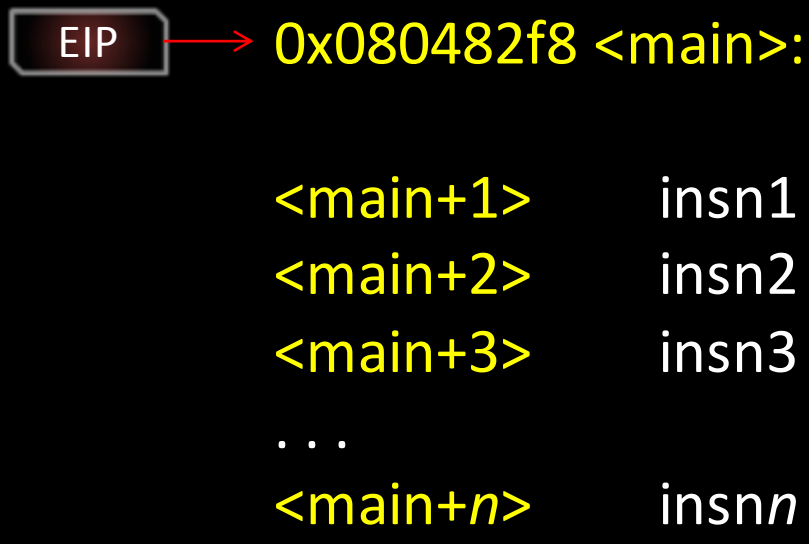
ATTACKER HAS FULL CONTROL OVER THE STACK

Execution Flow

In a program's normal execution

- EIP fetches instructions from memory
- EIP increments
- EIP fetches next instruction

Linear execution most of the time

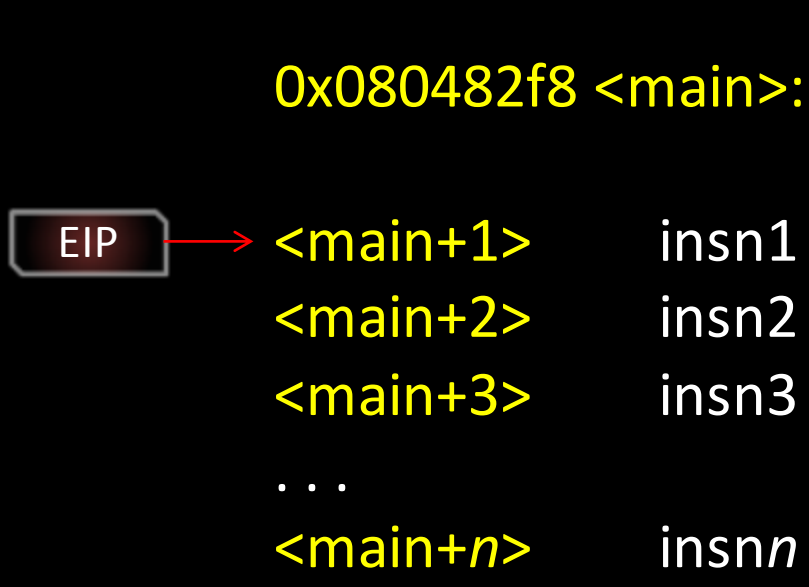


Execution Flow

In a program's normal execution

- EIP fetches instructions from memory
- EIP increments
- EIP fetches next instruction

Linear execution most of the time

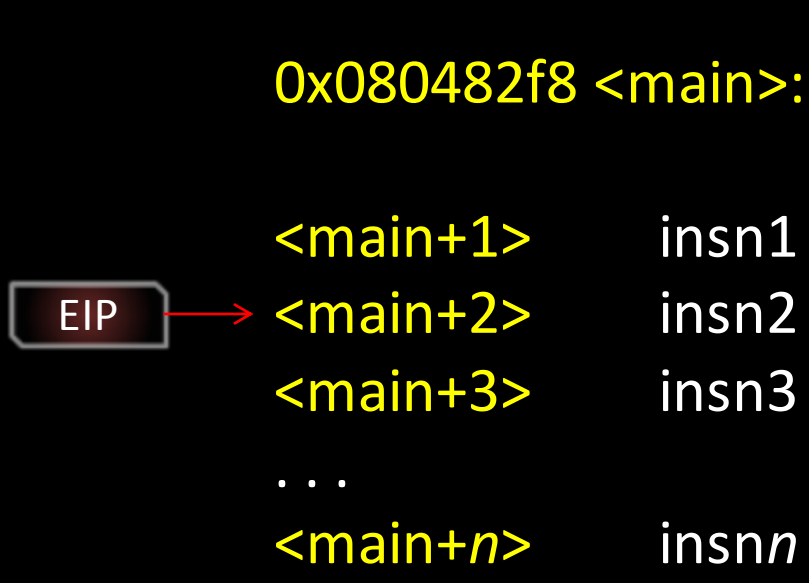


Execution Flow

In a program's normal execution

- EIP fetches instructions from memory
- EIP increments
- EIP fetches next instruction

Linear execution most of the time

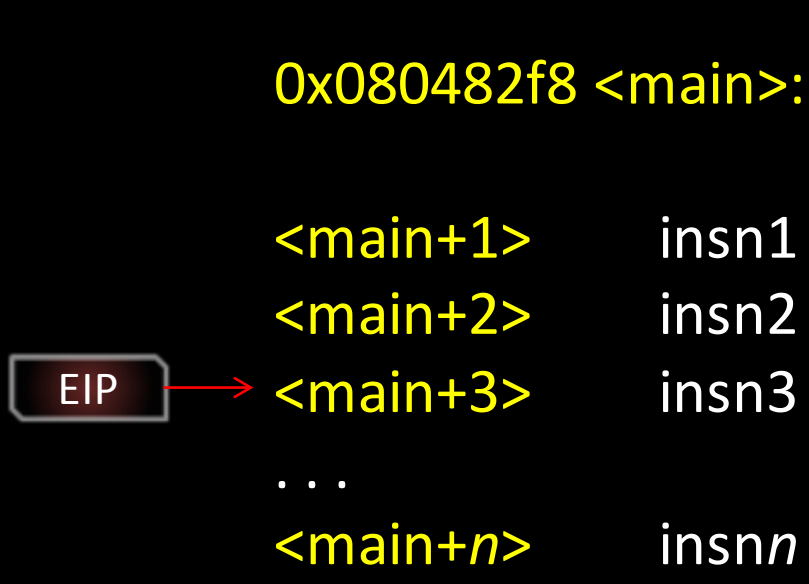


Execution Flow

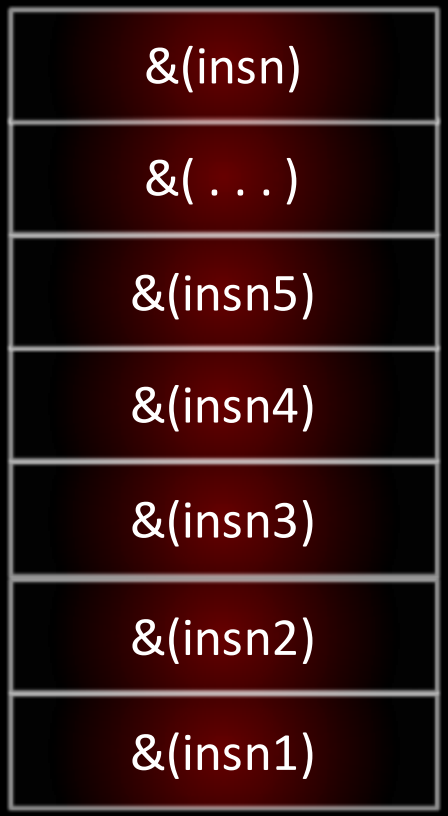
In a program's normal execution

- EIP fetches instructions from memory
- EIP increments
- EIP fetches next instruction

Linear execution most of the time



Execution Flow



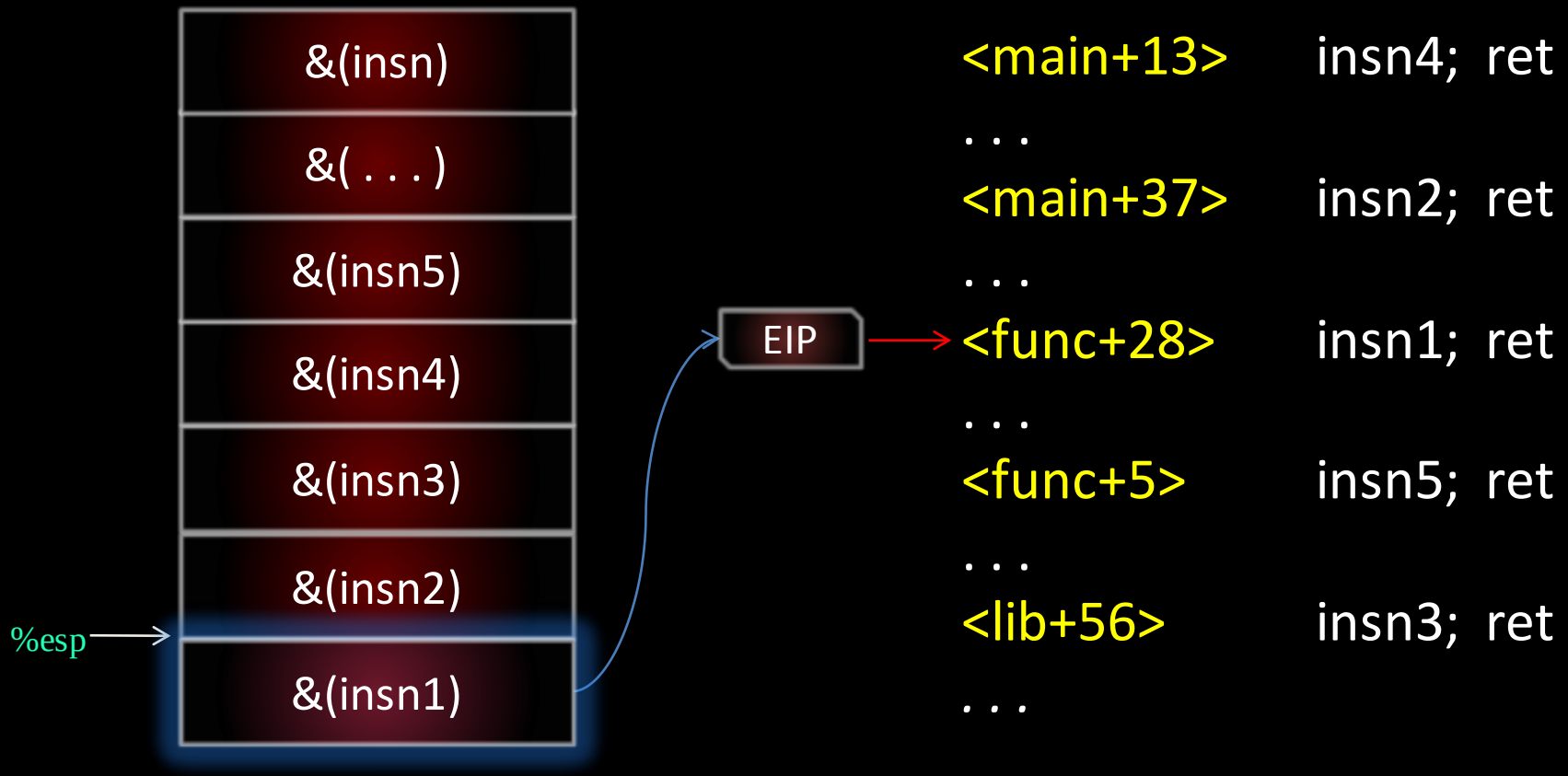
In Return Oriented Programming:

- The stack is overflowed with the address of many short blocks of instructions that end with ret
- The ret in the instruction makes EIP jump to the next address in the stack

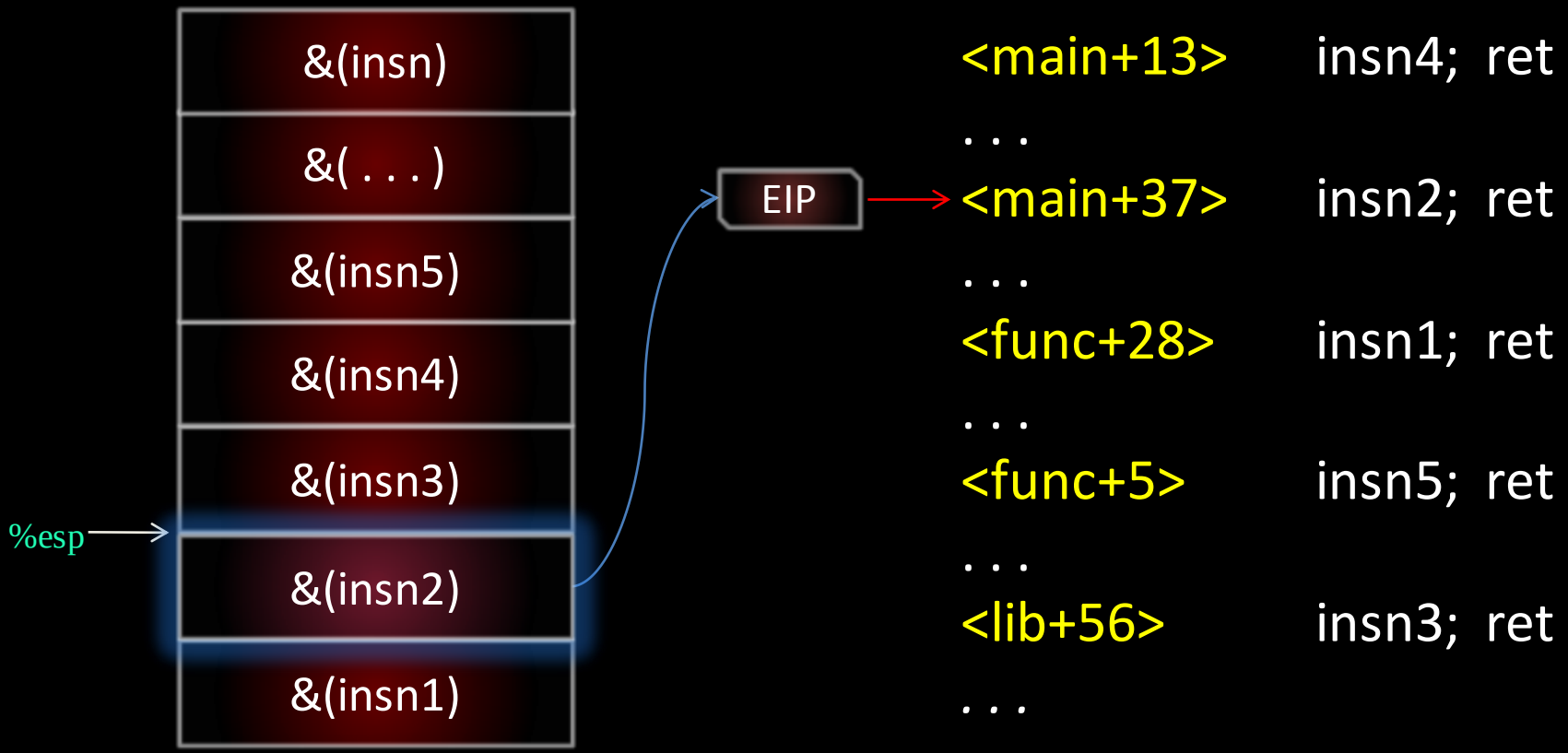
Execution Flow


```
<main+13>     insn4; ret
. . .
<main+37>     insn2; ret
. . .
<func+28>     insn1; ret
. . .
<func+5>      insn5; ret
. . .
<lib+56>      insn3; ret
. . .
```

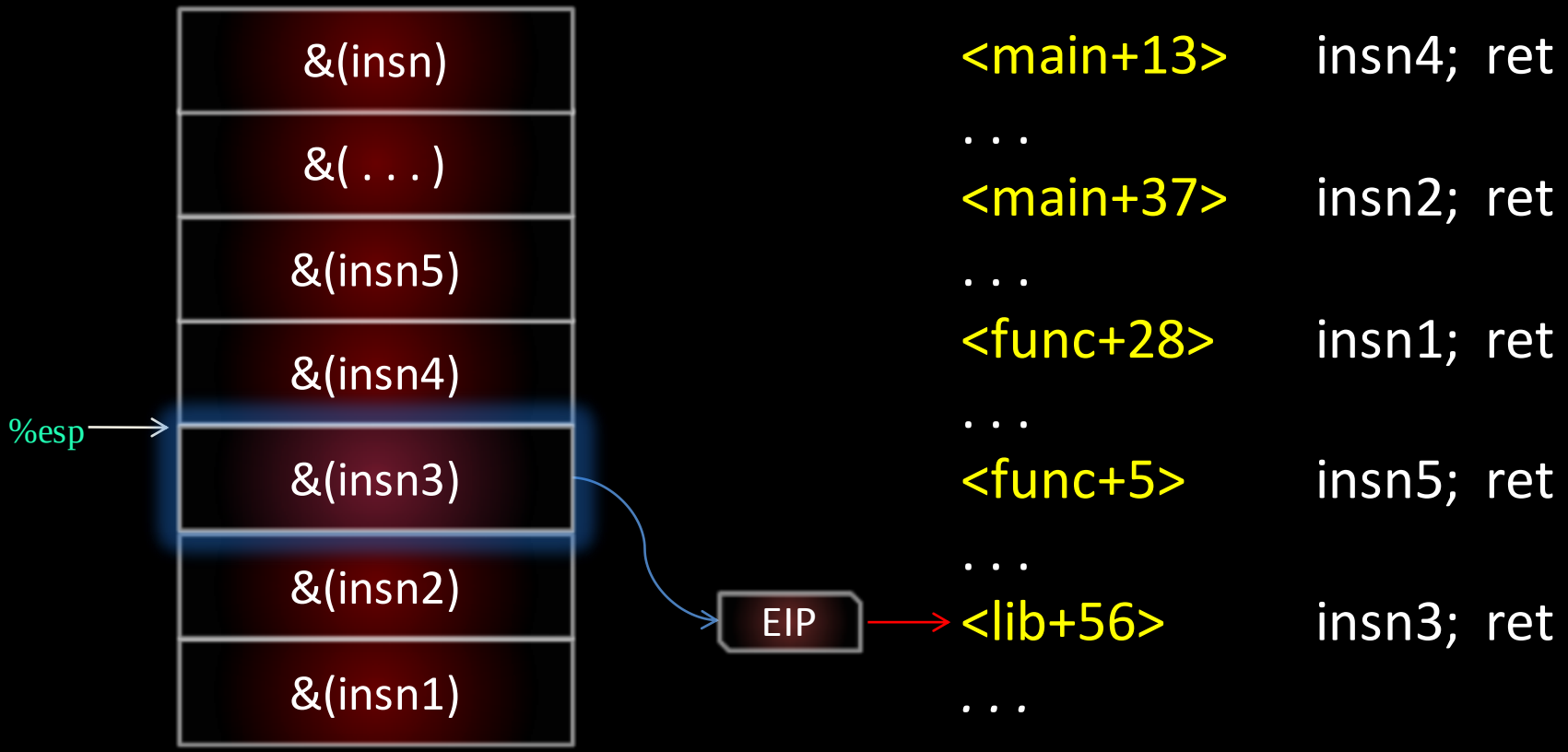
Execution Flow



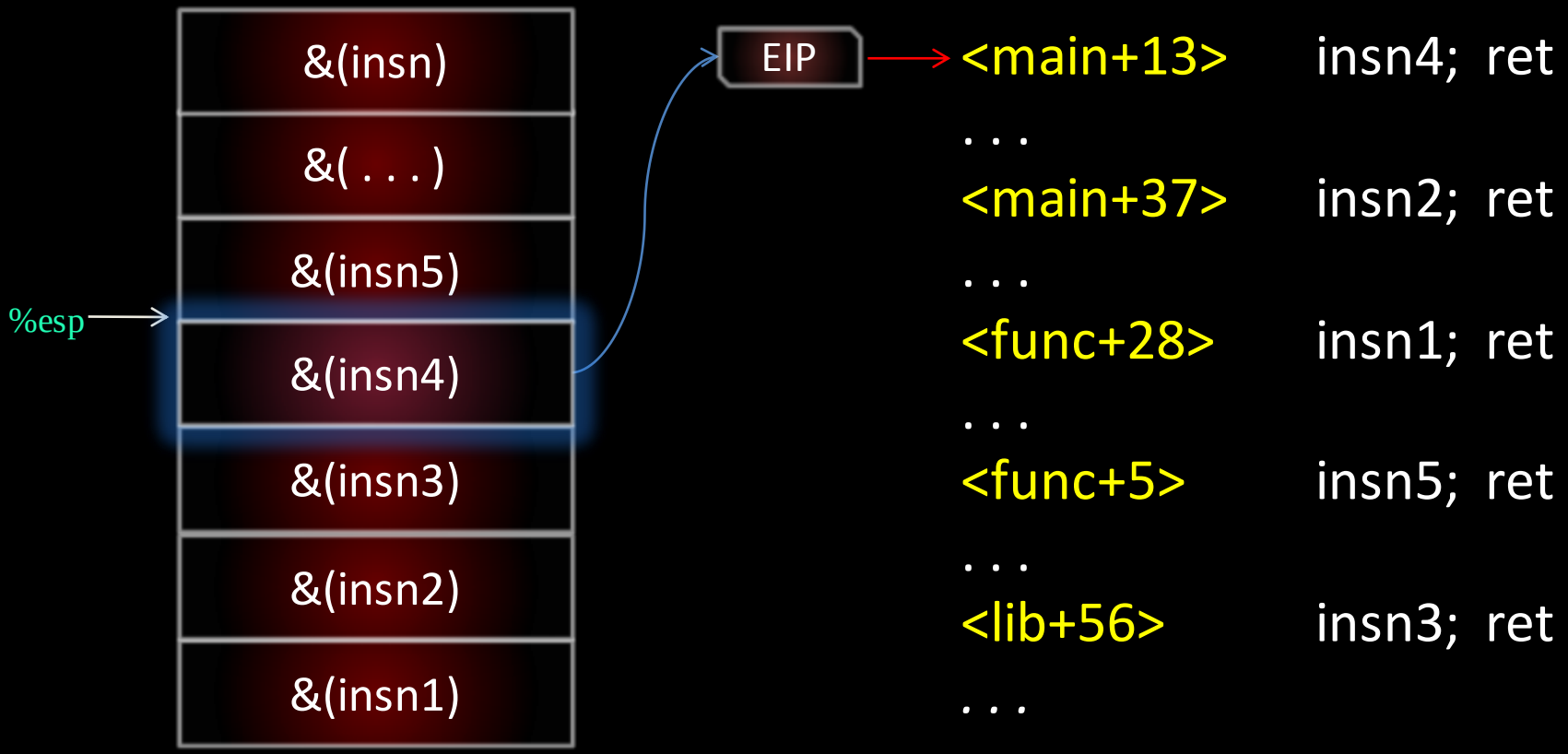
Execution Flow



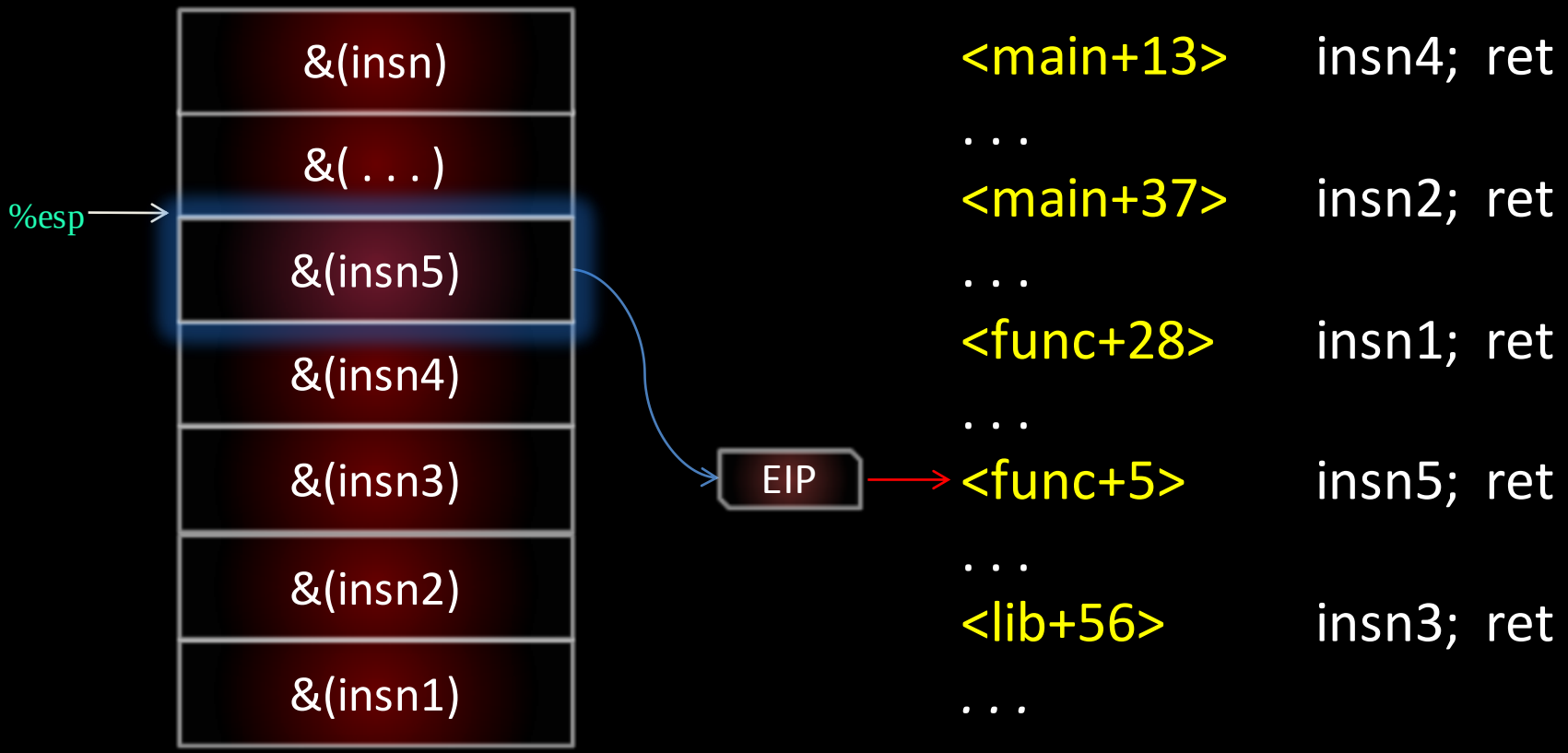
Execution Flow



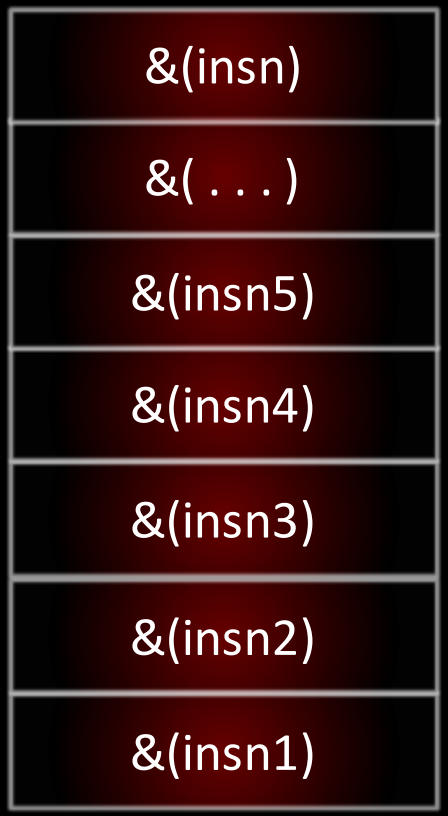
Execution Flow



Execution Flow



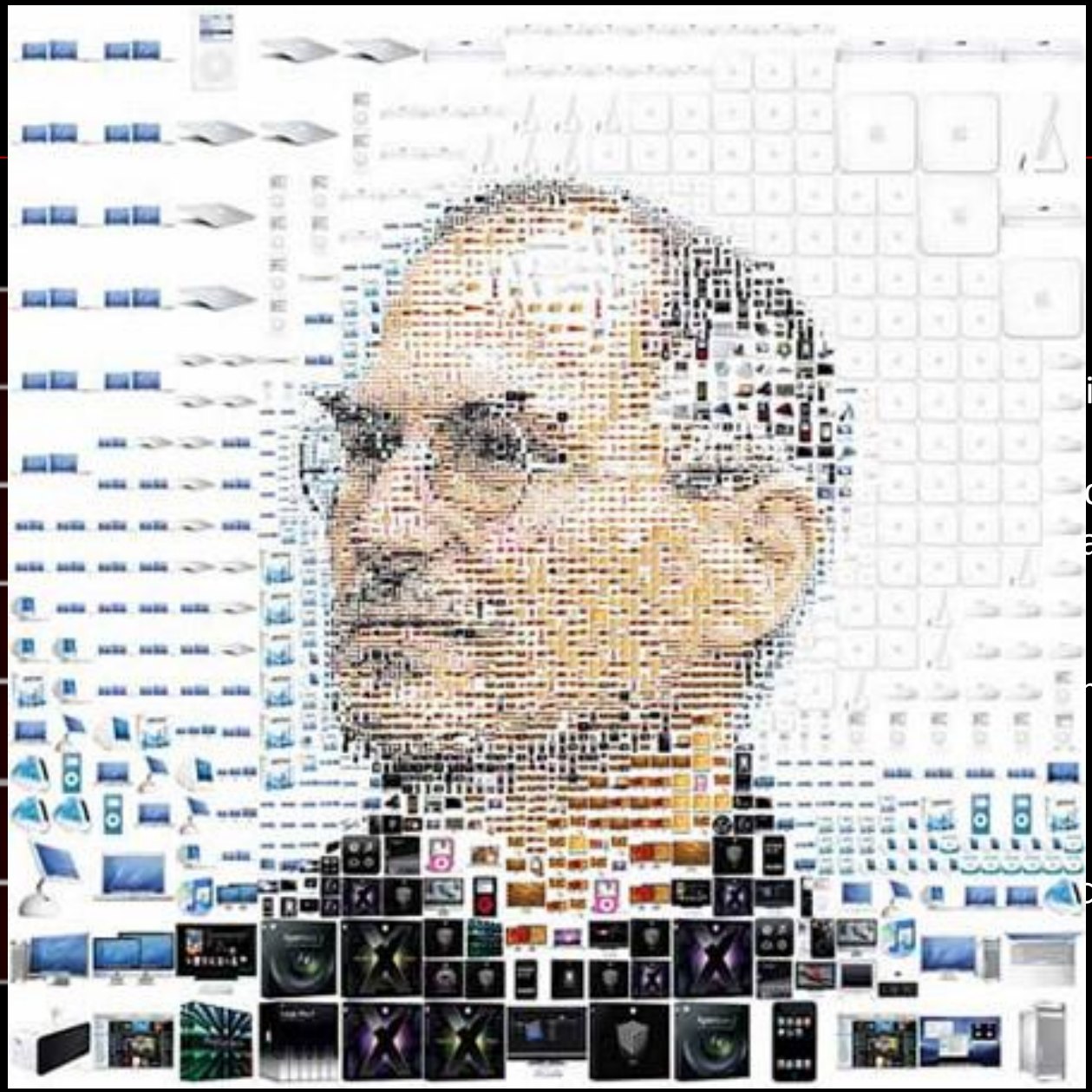
Execution Flow



In Return Oriented Programming:

- Small instructions sets are called one after another to simulate custom code execution.
- It is very probable to find in a binary all the instructions needed for any computation.

These instruction chunks are called **GADGETS**



ing:

called
ate

n a binary
for any

called

ing:

called
ate

n a binary
for any

called

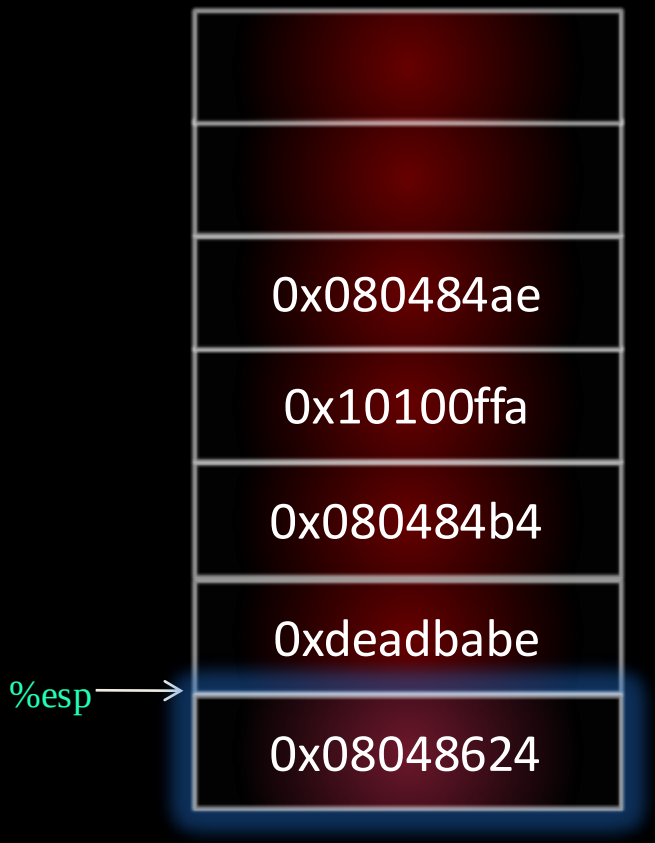
Example: Write word in memory

Write ‘0xdeadbabe’ to address 0x10101010


```
0x08048624:
    pop    %edx
    ret

0x080484b4:
    pop    %eax
    ret

0x0804a4ae:
    movl   %edx, 0x16(%eax)
    ret
```

Example: Write word in memory

Write ‘0xdeadbabe’ to address 0x10101010

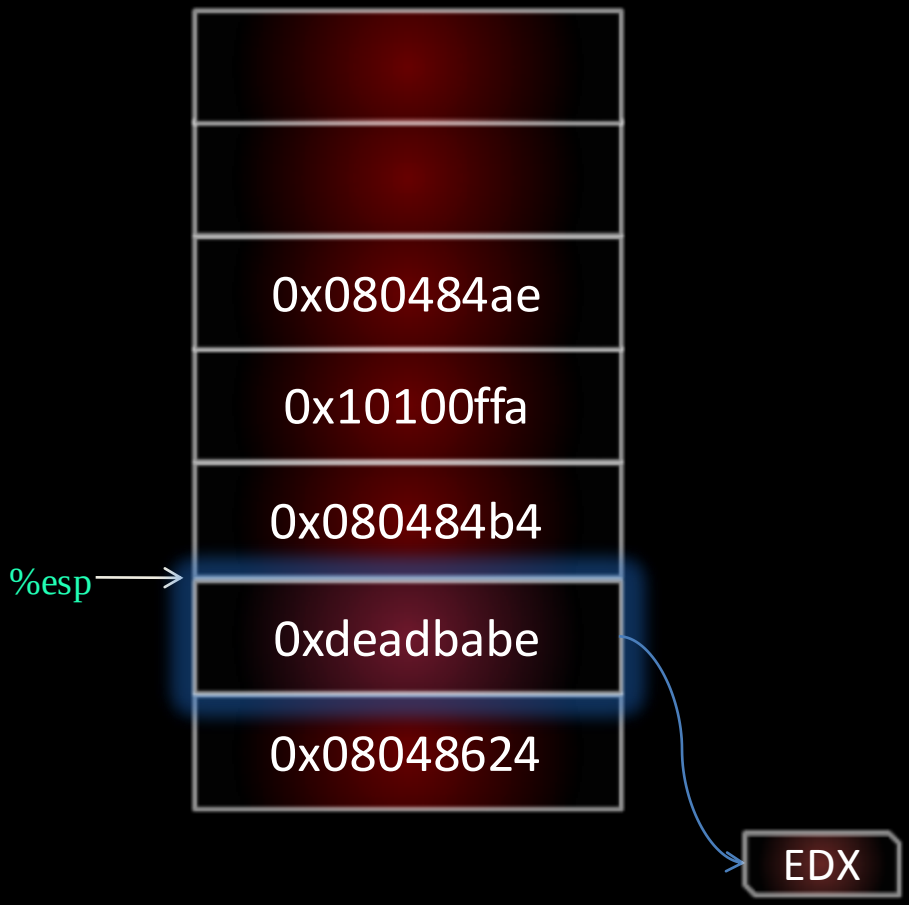


// Load a constant : movl $0xdeadbabe, %edx

```
0x08048624:
    pop    %edx
    ret

0x080484b4:
    pop    %eax
    ret

0x0804a4ae:
    movl   %edx, 0x16(%eax)
    ret
```

Example: Write word in memory

Write ‘0xdeadbabe’ to address 0x10101010

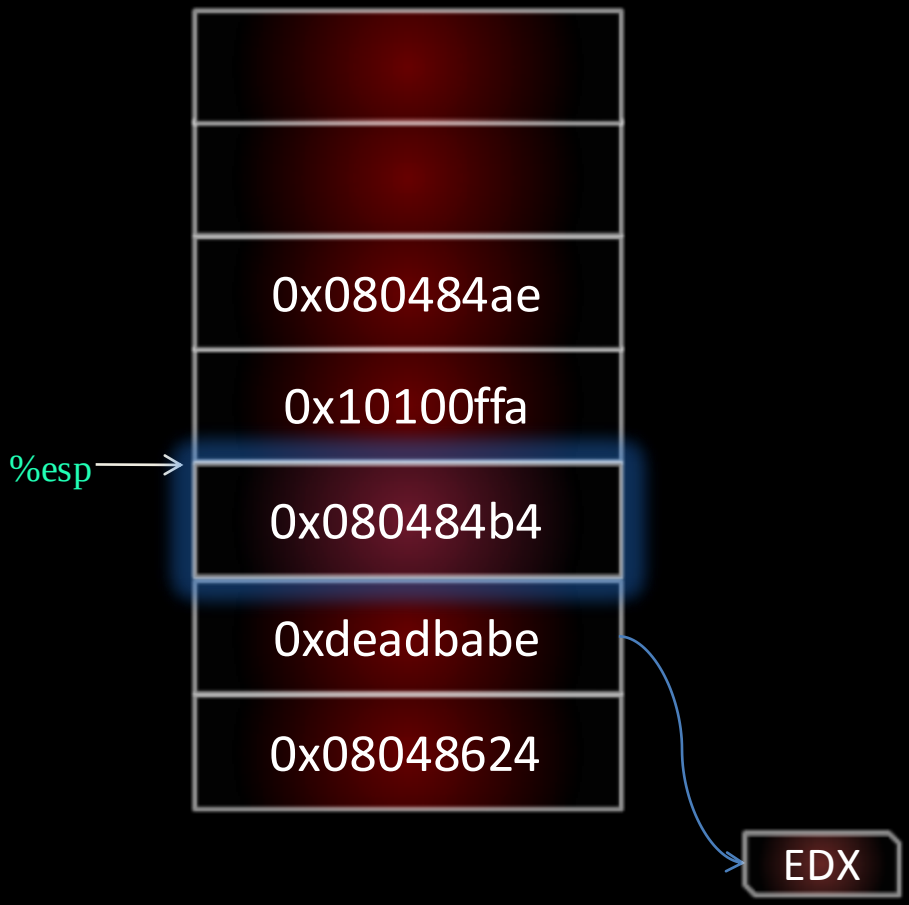


// Load a constant : movl $0xdeadbabe, %edx

```
0x08048624:
   pop   %edx
   ret

0x080484b4:
   pop   %eax
   ret

0x0804a4ae:
   movl  %edx, 0x16(%eax)
   ret
```

Example: Write word in memory

Write ‘0xdeadbabe’ to address 0x10101010



// Load a constant : movl $0xdeadbabe, %edx

```
0x08048624:
    pop    %edx
    ret

0x080484b4:
    pop    %eax
    ret

0x0804a4ae:
    movl   %edx, 0x16(%eax)
    ret
```

Example: Write word in memory

Write ‘0xdeadbabe’ to address 0x10101010


```
0x08048624:
    pop    %edx
    ret
// Load a constant :  movl $0x10100ffa, %eax
0x080484b4:
    pop    %eax
    ret

0x0804a4ae:
    movl   %edx, 0x16(%eax)
    ret
```

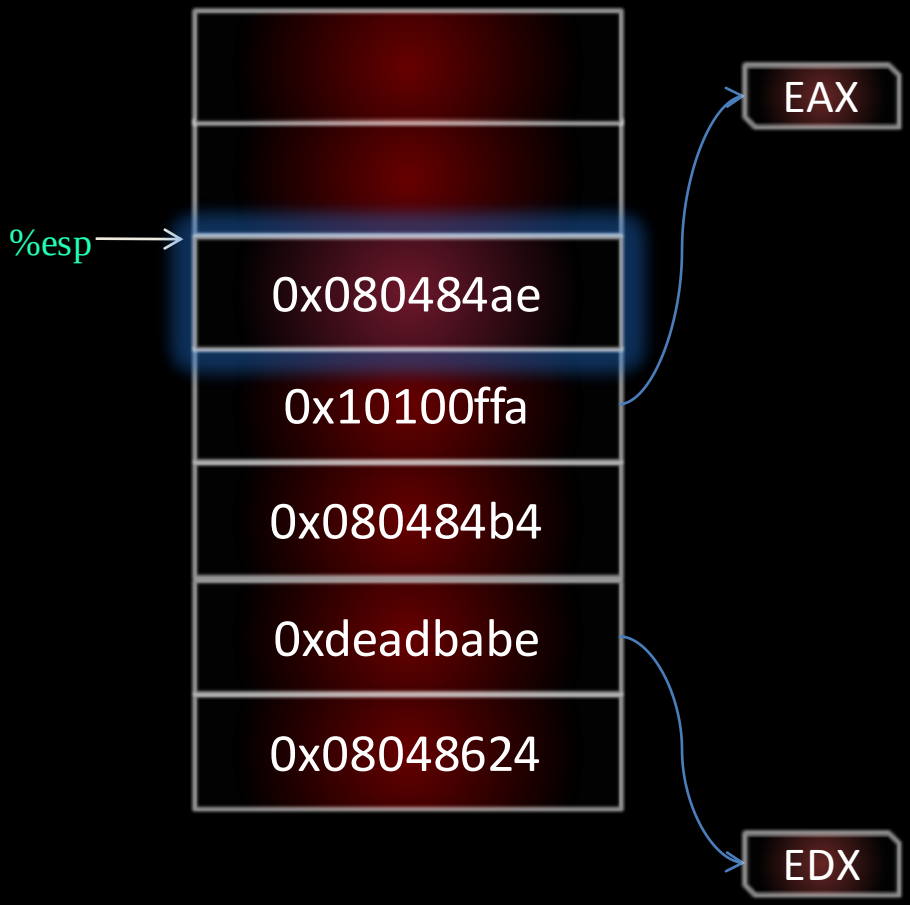
Example: Write word in memory

Write ‘0xdeadbabe’ to address 0x10101010


```
0x08048624:
    pop    %edx
    ret
// Load a constant :  movl $0x10100ffa, %eax
0x080484b4:
    pop    %eax
    ret

0x0804a4ae:
    movl   %edx, 0x16(%eax)
    ret
```

Example: Write word in memory

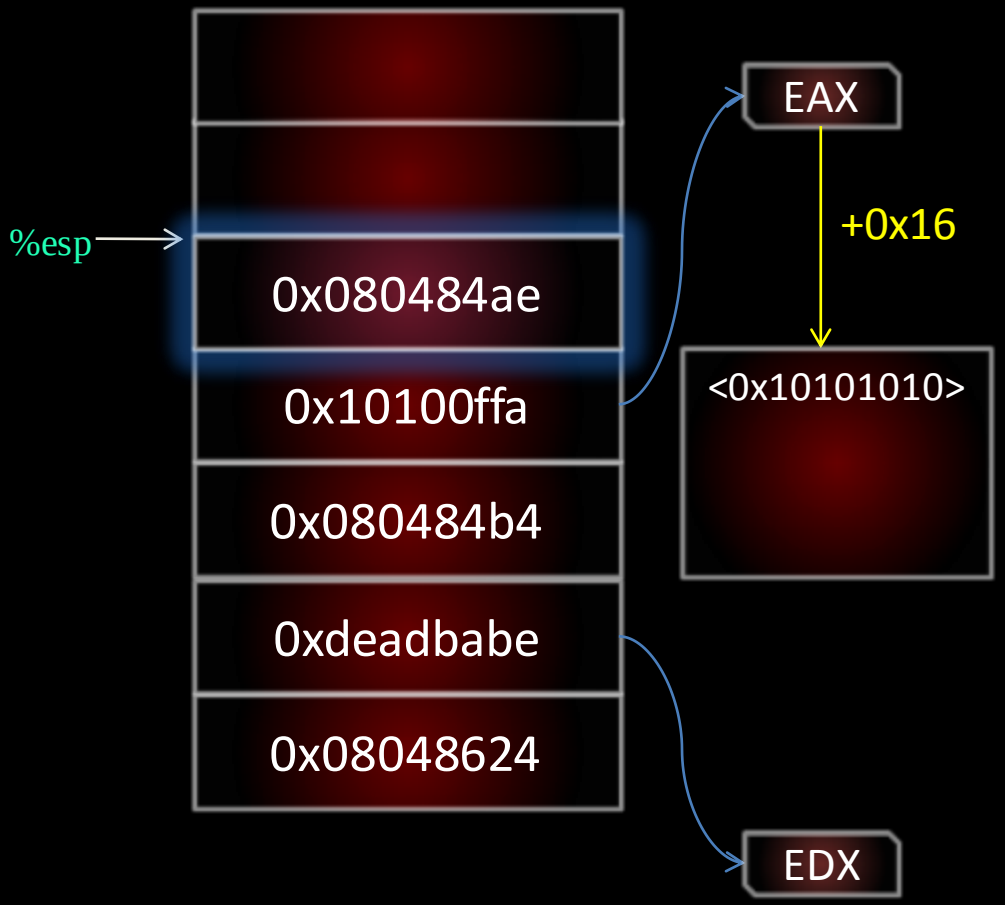
Write ‘0xdeadbabe’ to address 0x10101010




```
0x08048624:
    pop    %edx
    ret

0x080484b4:
    pop    %eax
    ret
// Store in memory
0x0804a4ae:
    movl   %edx, 0x16(%eax)
    ret
```

Example: Write word in memory

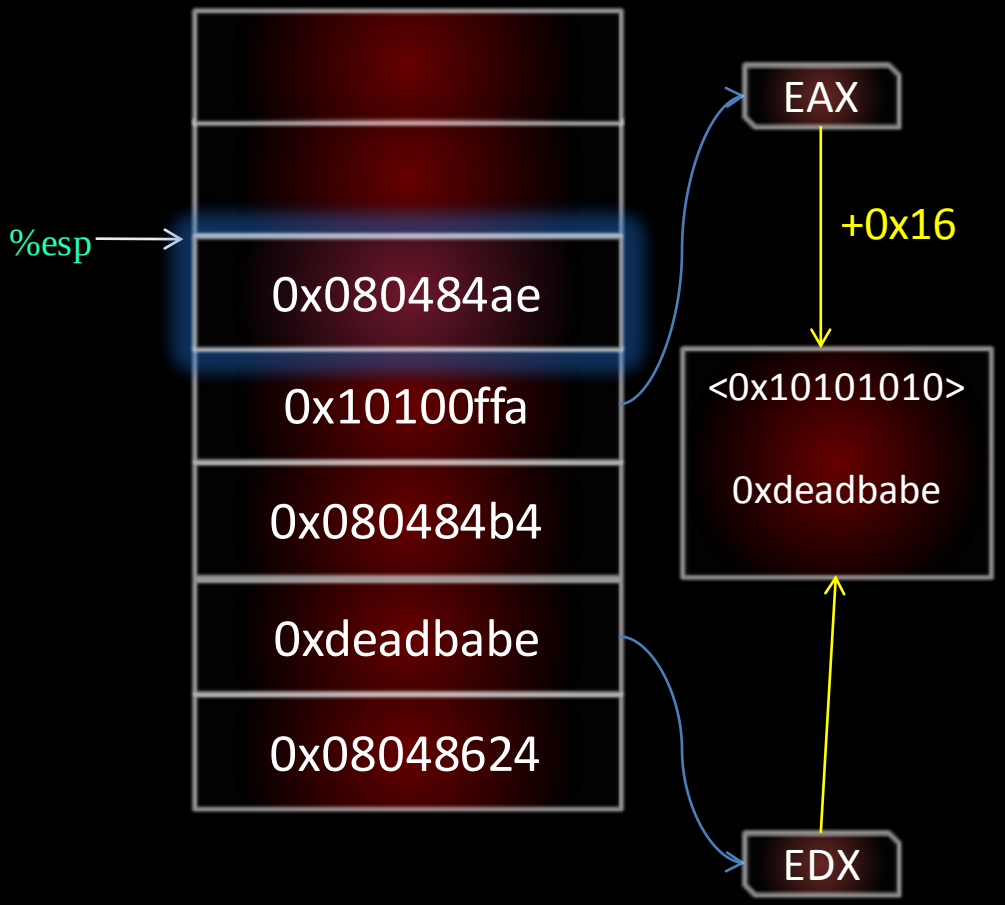
Write ‘0xdeadbabe’ to address 0x10101010




```
0x08048624:
    pop   %edx
    ret

0x080484b4:
    pop   %eax
    ret
// Store in memory
0x0804a4ae:
    movl  %edx, 0x16(%eax)
    ret
```

Example: Write word in memory

Write ‘0xdeadbabe’ to address 0x10101010




```
0x08048624:
   pop   %edx
   ret

0x080484b4:
   pop   %eax
   ret

// Store in memory
0x0804a4ae:
   movl  %edx, 0x16(%eax)
   ret
```

Finding useful instructions

- Scan executable for RET instructions
  OpCode (0xc3)
- When RET is found, look backwards
- Make catalog of code chunks and addresses

GALILEO algorithm, *`The Geomety of Innocent Flesh in the Bone', Hovav Shacham*

Exploiting payloads

Windows

Linux

Exploiting payloads

Windows

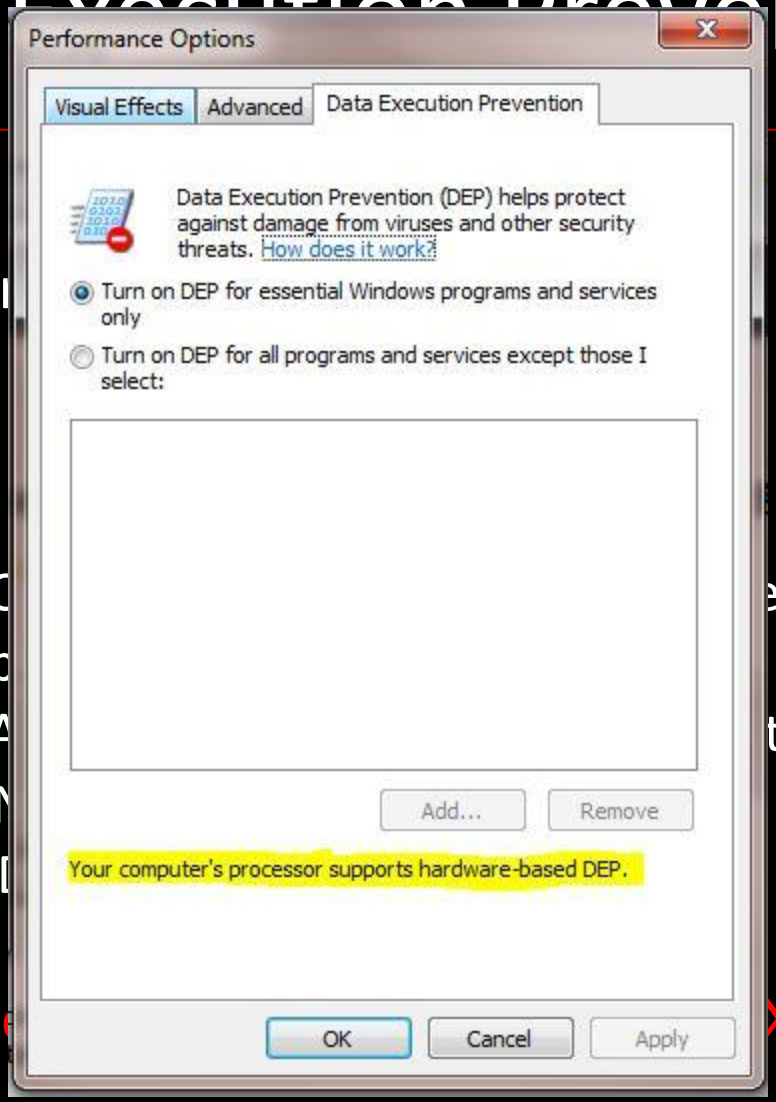
Linux

Data Execution Prevention

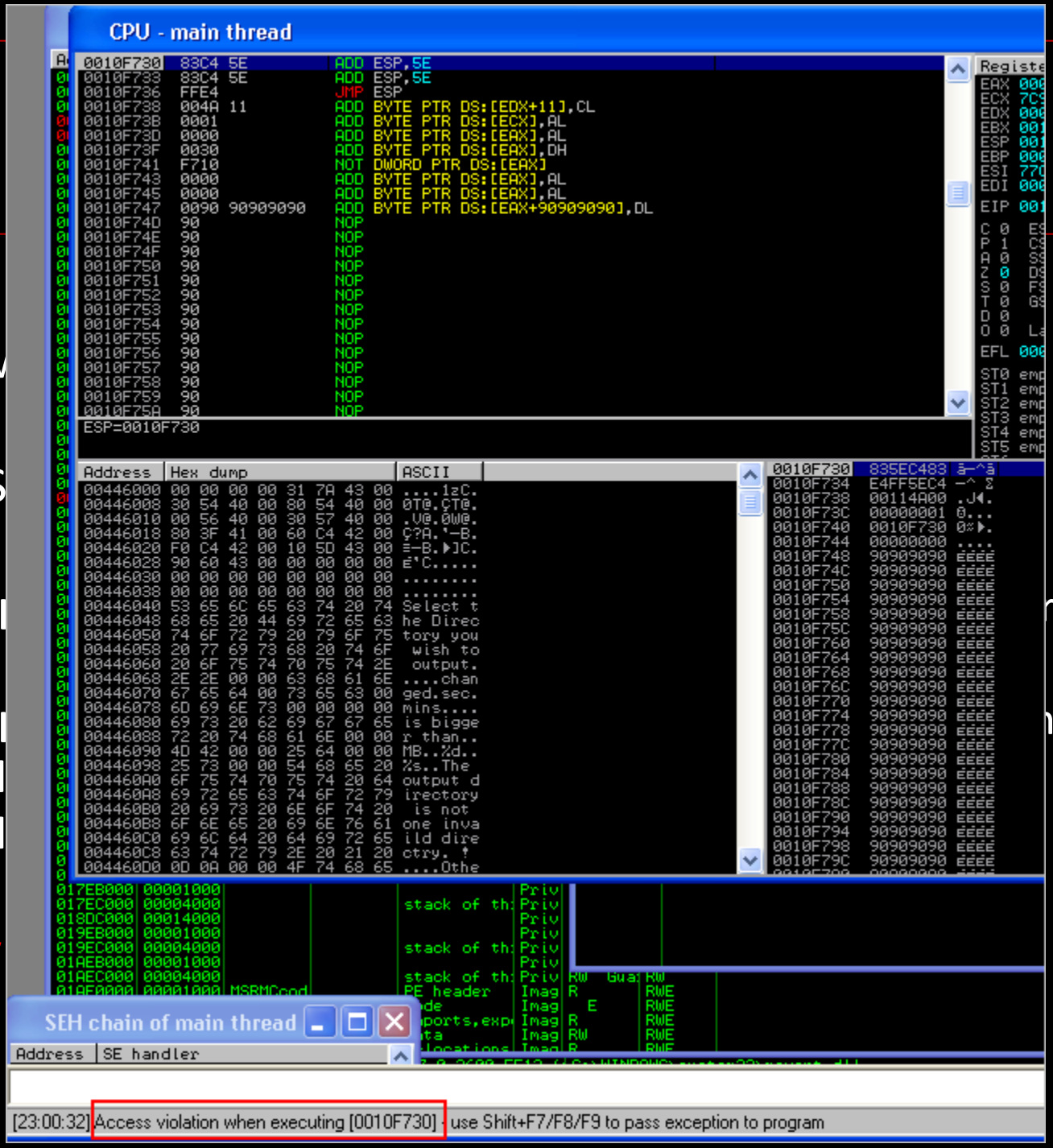
- Hardware DEP in Win32 uses NX/XD bit
- DEP Settings:
  - **OptIn:** Critical programs protected by default, other programs if supported.
  - **OptOut:** All programs protected by default (exception list)
  - **AlwaysOn:** No exceptions
  - **AlwaysOff**: DEP is turned off.
- Every module needs to be compiled with /NXCOMPAT flag

Data Execution Prevention

- Hardware DEP i
- DEP Settings:
  - **OptIn:** C efault, other p
  - **OptOut:** A t (exception list)
  - **AlwaysOn:** N
  - **AlwaysOff:** D
- Every module ne XCOMPAT flag



- Hardv
- DEP S
  - O
  - O
  - Al
  - Al
- Every



list)

flag

Data Execution Prevention

- DEP could be turned off for the whole process by calling the function `NtSetInformationProcess()`
- «Permanent DEP» introduced in XP SP3 & Vista SP1, prevents DEP from being disabled.
- Windows OSs default settings:
  - **XP SP1-SP2, Vista SP0:** OptIn
  - **XP SP3, Vista SP1:** OptIn + Permanent DEP
  - **Windows 7:** Optin + Permanent DEP
  - **Server 2003 SP1:** OptOut
  - **Server 2008:** OptOut + Permament DEP

Data Execution Prevention

- Potencial DEP bypasses:
  - `NtSetInformationProcess()`
  - `SetProcessDepPolicy()`
  - `VirtualProtect(PAGE_READ_WRITE_EXECUTABLE)`
  - `HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc()`
  - `VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE`
  - `Pure Return Oriented Programming Shellcode`

Data Execution Prevention

- Potential DEP bypasses:
  - `NtSetInformationProcess()`
  - `SetProcessDepPolicy()`
  - `VirtualProtect(PAGE_READ_WRITE_EXECUTABLE)`
  - `HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc()`
  - `VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE)`
  - `Pure Return Oriented Programming Shellcode`

What about ASLR?

- You need to know the address of the gadgets for ROP.

DEP + ASLR = Hard Exploitation

- **Search for a module that doesn't opt-in for ASLR, then use the instructions from that module.**
- **Find a memory-layout disclosure bug.**

Exploiting payloads

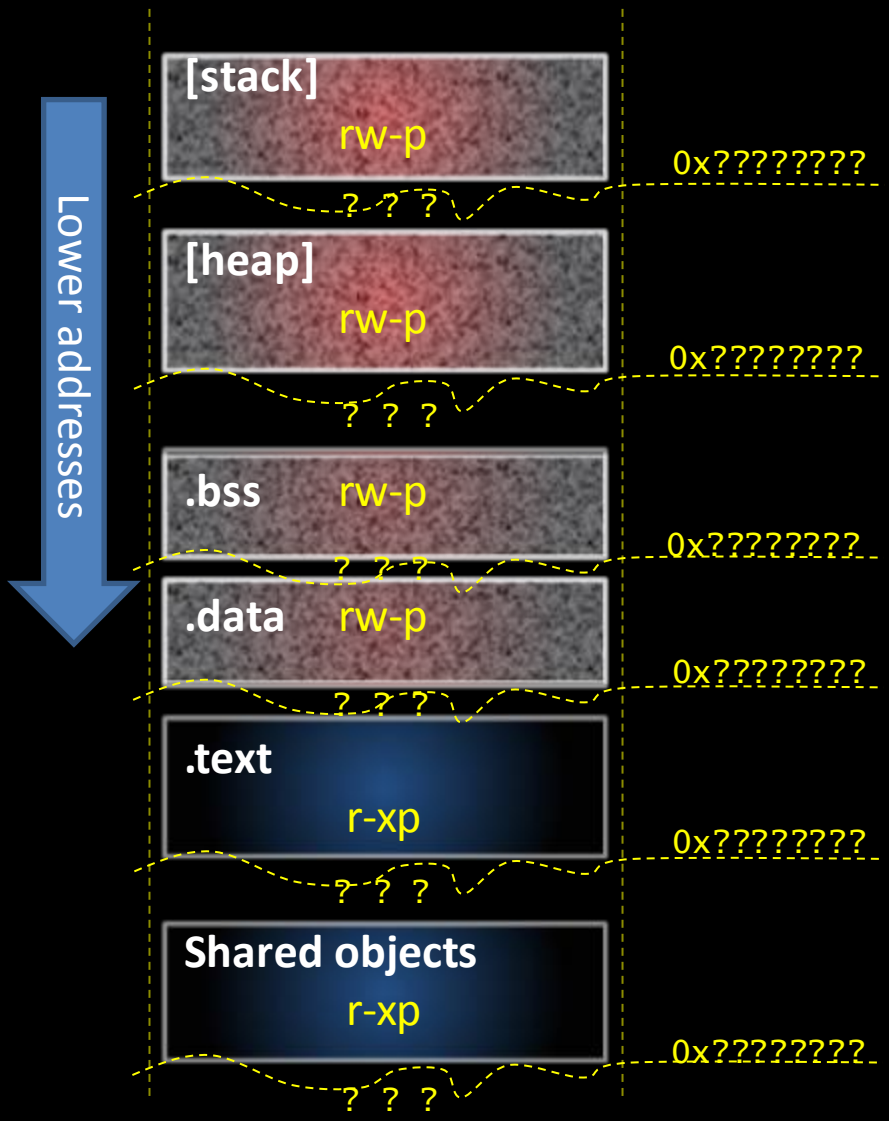
Windows

Linux

Linux NX Memory

- NX Memory can be implemented through different kernel patches
  - GrSecurity
  - PaX
  - ExecShield
- Many distros implement NX Memory
  - Ubuntu
  - Fedora (ExecShield)
  - Hardened Gentoo (Gr Security)
- Protection level varies depending in the implemented patch

Linux ASLR

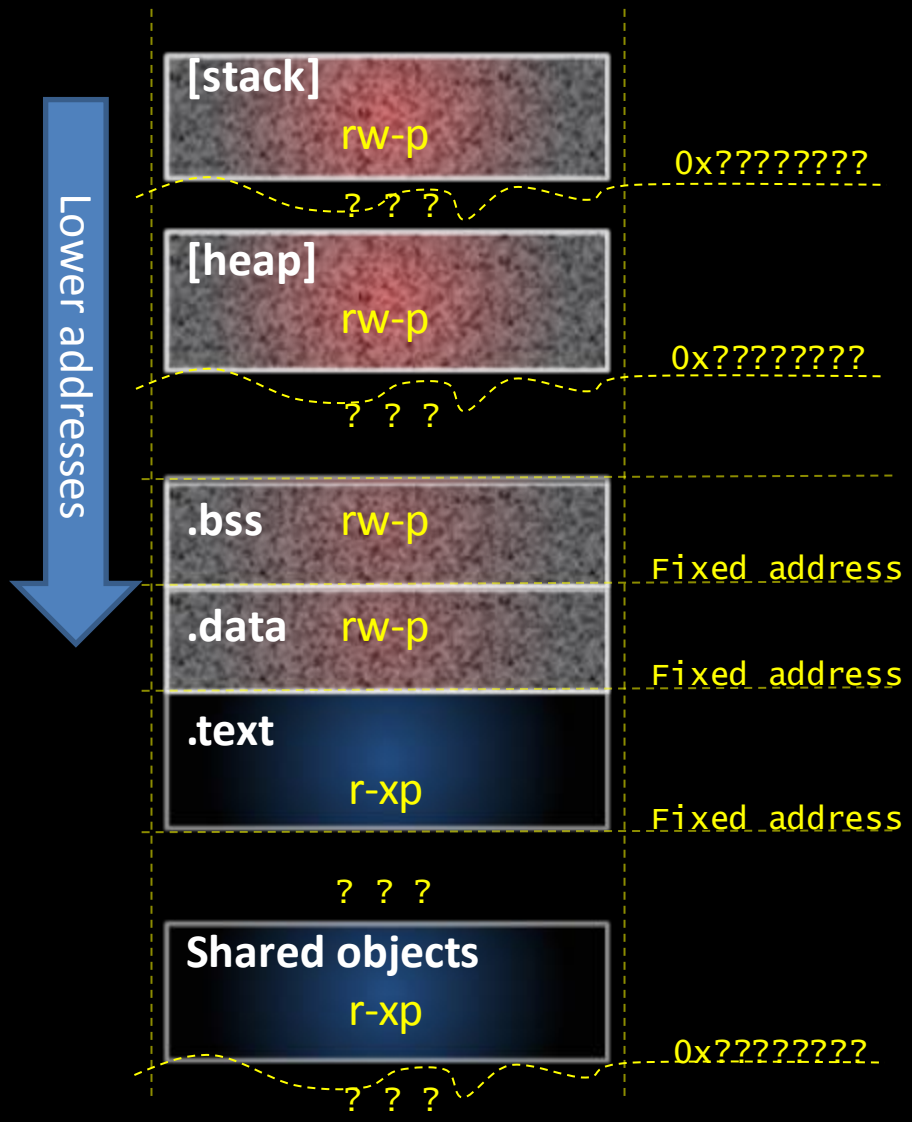


Introduced to **Linux** by PaX in 2001

Kernel has an ASLR implementation via randomize_va_space

Full ASLR? Not quite ;)

Linux ASLR

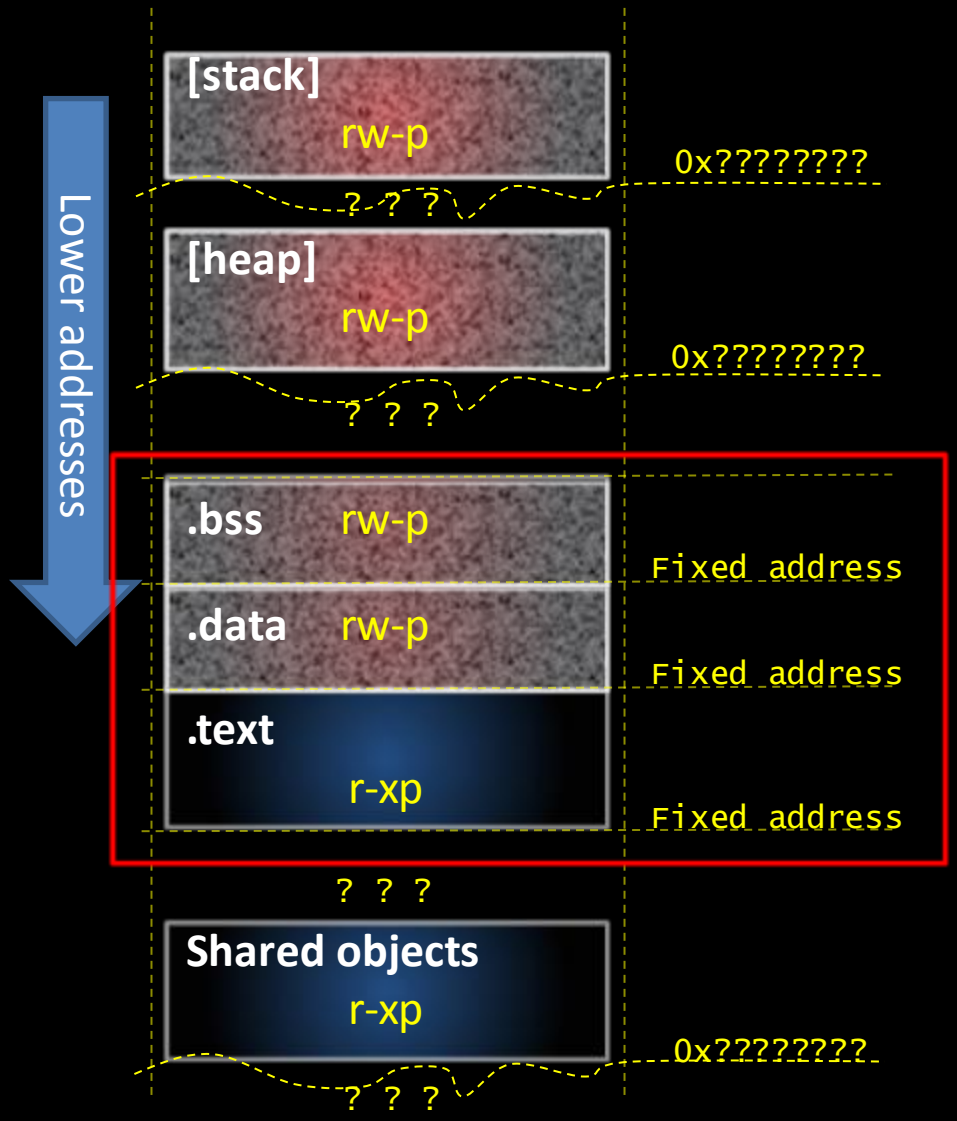


Full memory randomization in the process only when the program was compiled to be **Position Independent Executable.**

Otherwise, the exec base is not randomized. So not 100% ASLR

No-PIE code is still very common in most linux distros. (PIE decreases performance)

Linux ASLR



Full memory randomization in the process only when the program was compiled to be **Position Independent Executable.**

Otherwise, the exec base is not randomized. So not 100% ASLR

No-PIE code is still very common in most linux distros.

```
trew@DarkLight ~ $ readelf -S vuln
There are 30 section headers, starting at offset 0x92c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0804818c 00018c 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481ac 0001ac 0000b0 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804825c 00025c 000073 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          080482d0 0002d0 000016 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         080482e8 0002e8 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048308 000308 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048310 000310 000048 08   A  5  12  4
  [11] .init             PROGBITS        08048358 000358 000030 00  AX  0   0  4
  [12] .plt              PROGBITS        08048388 000388 0000a0 04  AX  0   0  4
  [13] .text             PROGBITS        08048430 000430 0001dc 00  AX  0   0 16
  [14] .fini             PROGBITS        0804860c 00060c 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048628 000628 000028 00   A  0   0  4
  [16] .eh_frame_hdr     PROGBITS        08048650 000650 000024 00   A  0   0  4
  [17] .eh_frame         PROGBITS        08048674 000674 00007c 00   A  0   0  4
  [18] .ctors            PROGBITS        080496f0 0006f0 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        080496f8 0006f8 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049700 000700 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049704 000704 0000c8 08  WA  6   0  4
  [22] .got              PROGBITS        080497cc 0007cc 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        080497d0 0007d0 000030 04  WA  0   0  4
  [24] .data             PROGBITS        08049800 000800 000004 00  WA  0   0  4
  [25] .bss              NOBITS          08049804 000804 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 000804 00002c 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 000830 0000fc 00      0   0  1
  [28] .symtab           SYMTAB          00000000 000ddc 000470 10     29  45  4
  [29] .strtab           STRTAB          00000000 00124c 000265 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
```

Lower addresses

ation in
the
to be

se is not
0% ASLR

distros.

? ? ?

Linux Overflows Exploitation

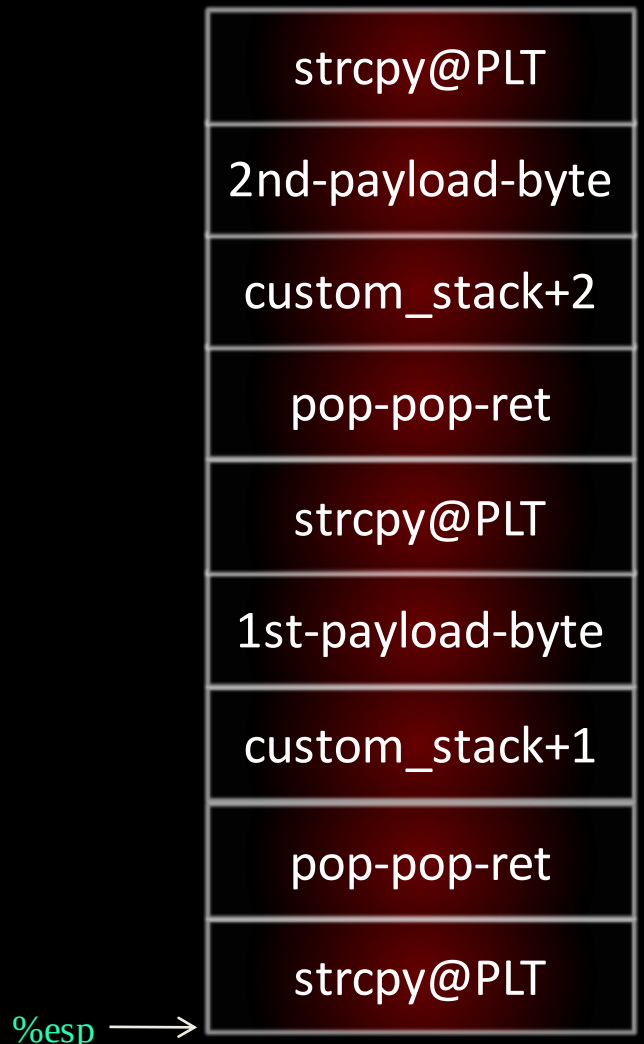
- NX Memory + ASLR (w/out PIE) + AAAS bypass.
- Long Le's Technique '*Data Reuse for ROP Exploits*'

- 2 Stage attack:
  - Stage-0: Transfer payload to custom stack at fixed location
  - Stage-1: Execute payload from custom stack as normal ROP

Sta

```
trew@DarkLight ~ $ readelf -S vuln
There are 30 section headers, starting at offset 0x92c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0804818c 00018c 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481ac 0001ac 0000b0 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804825c 00025c 000073 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          080482d0 0002d0 000016 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         080482e8 0002e8 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048308 000308 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048310 000310 000048 08   A  5  12  4
  [11] .init             PROGBITS        08048358 000358 000030 00  AX  0   0  4
  [12] .plt              PROGBITS        08048388 000388 0000a0 04  AX  0   0  4
  [13] .text             PROGBITS        08048430 000430 0001dc 00  AX  0   0 16
  [14] .fini             PROGBITS        0804860c 00060c 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048628 000628 000028 00   A  0   0  4
  [16] .eh_frame_hdr     PROGBITS        08048650 000650 000024 00   A  0   0  4
  [17] .eh_frame         PROGBITS        08048674 000674 00007c 00   A  0   0  4
  [18] .ctors            PROGBITS        080496f0 0006f0 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        080496f8 0006f8 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049700 000700 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049704 000704 0000c8 08  WA  6   0  4
  [22] .got              PROGBITS        080497cc 0007cc 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        080497d0 0007d0 000030 04  WA  0   0  4
  [24] .data             PROGBITS        08049800 000800 000004 00  WA  0   0  4
  [25] .bss              NOBITS          08049804 000804 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 000804 00002c 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 000830 0000fc 00      0   0  1
  [28] .symtab           SYMTAB          00000000 000ddc 000470 10     29  45  4
  [29] .strtab           STRTAB          00000000 00124c 000265 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
```

Stage-0: Transfer payload to custom stack

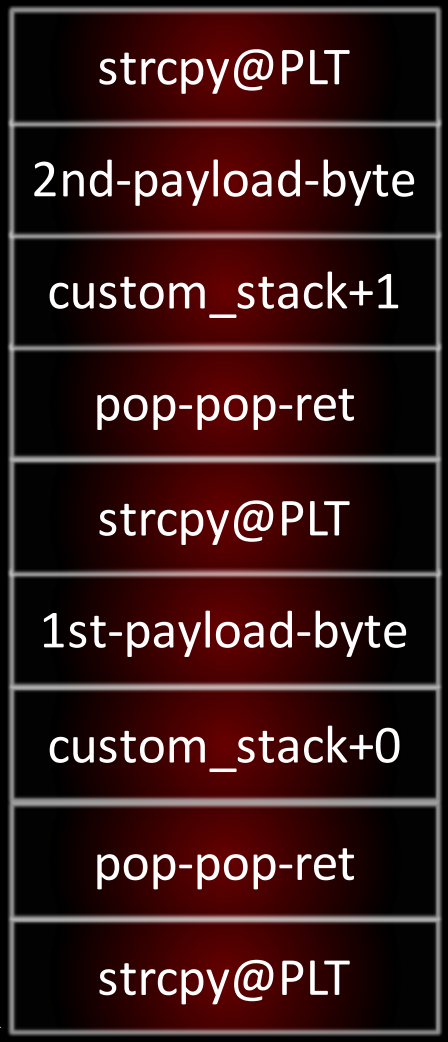


Search for individual bytes in binary to construct payload.

Copy with strcpy the bytes to custom stack, one-by-one.

Use Procedure Linkage Table to jump to strcpy.

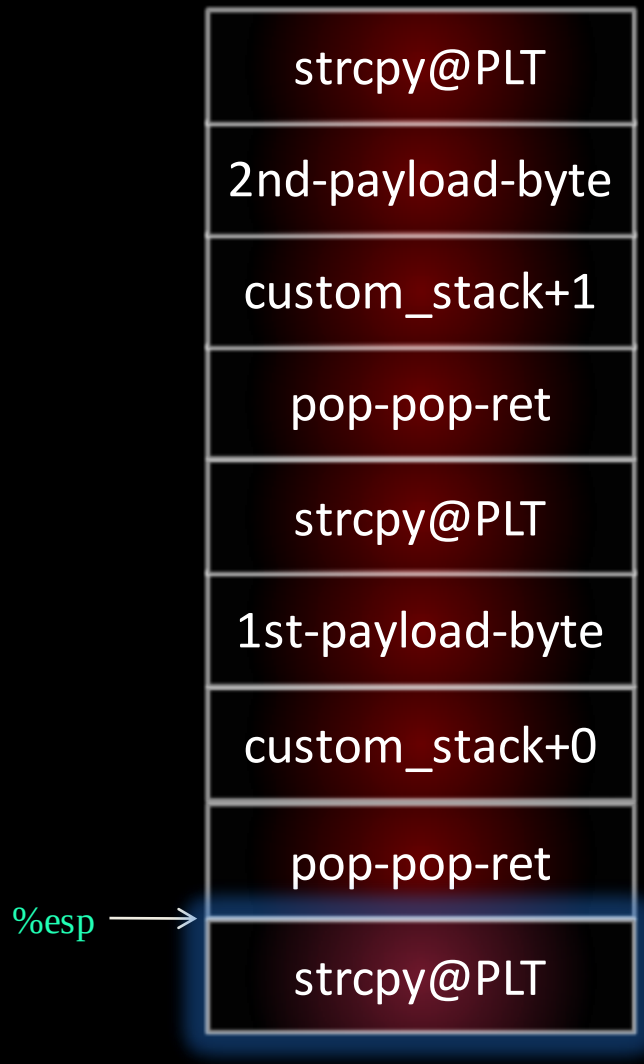
Stage-0: Transfer payload to custom stack



Payload = ['/bin/sh']

.data custom stack (fixed location)

Stage-0: Transfer payload to custom stack



Payload = ['/bin/sh']

```
<strcpy>:
. . . ; function prologue
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . . ; data copy instructions
ret
```

.data custom stack (fixed location)

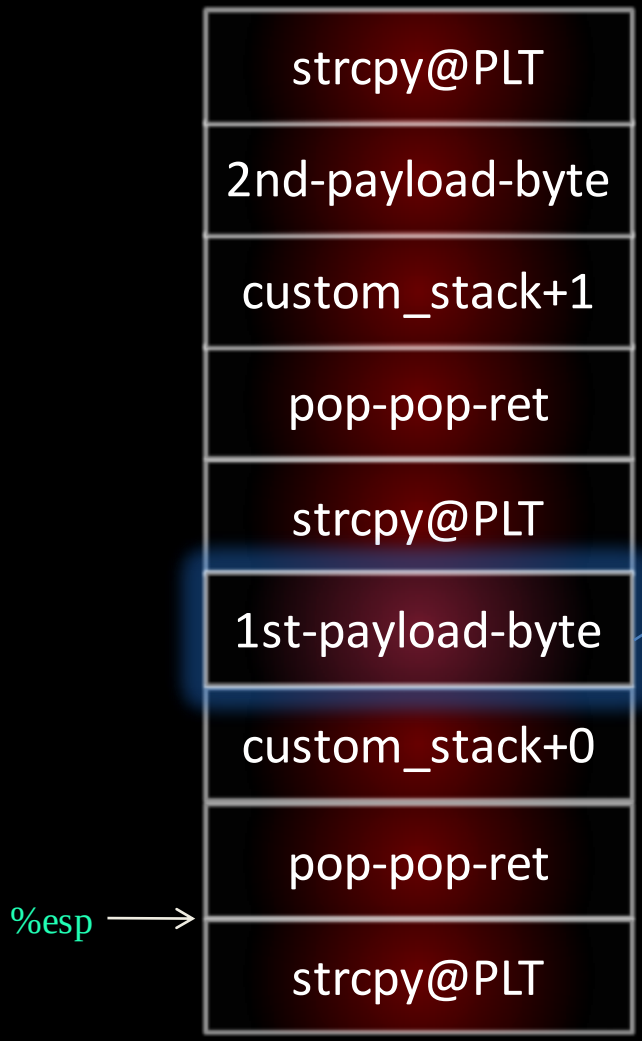
Stage-0: Transfer payload to custom stack

Payload = ['/bin/sh']

```
<strcpy>:
. . . ; function prologue
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . . ; data copy instructions
ret
```

.data custom stack (fixed location)

Stage-0: Transfer payload to custom stack



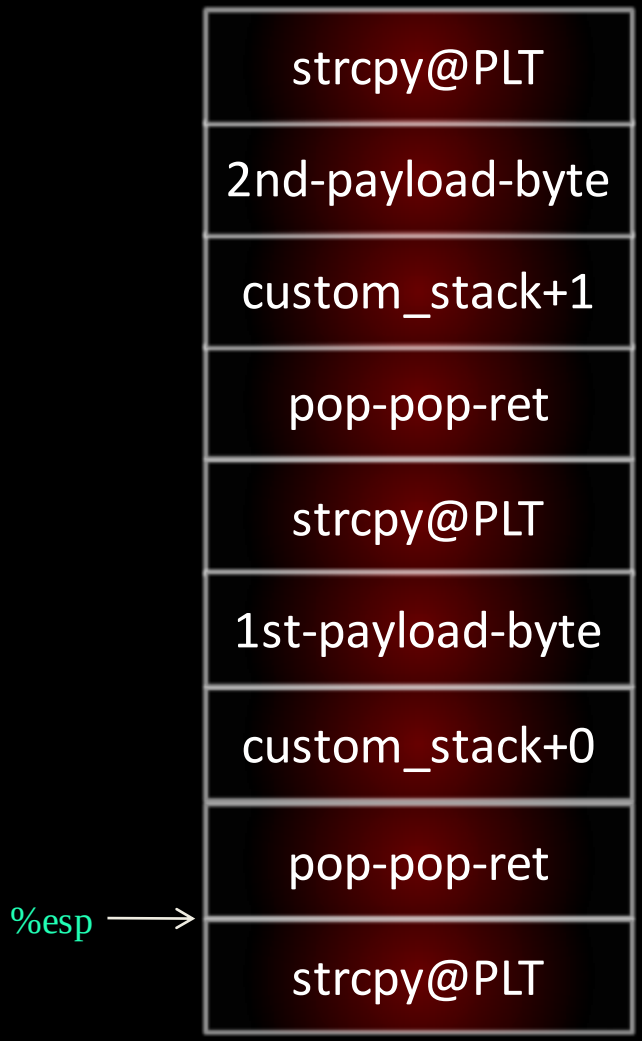
Payload = ['/bin/sh']

.text

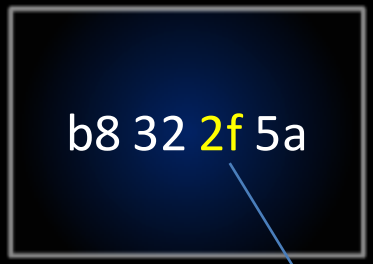
```
<strcpy>:
. . . ; function prologue
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . . ; data copy instructions
ret
```

.data custom stack (fixed location)

Stage-0: Transfer payload to custom stack



Payload = ['/bin/sh']

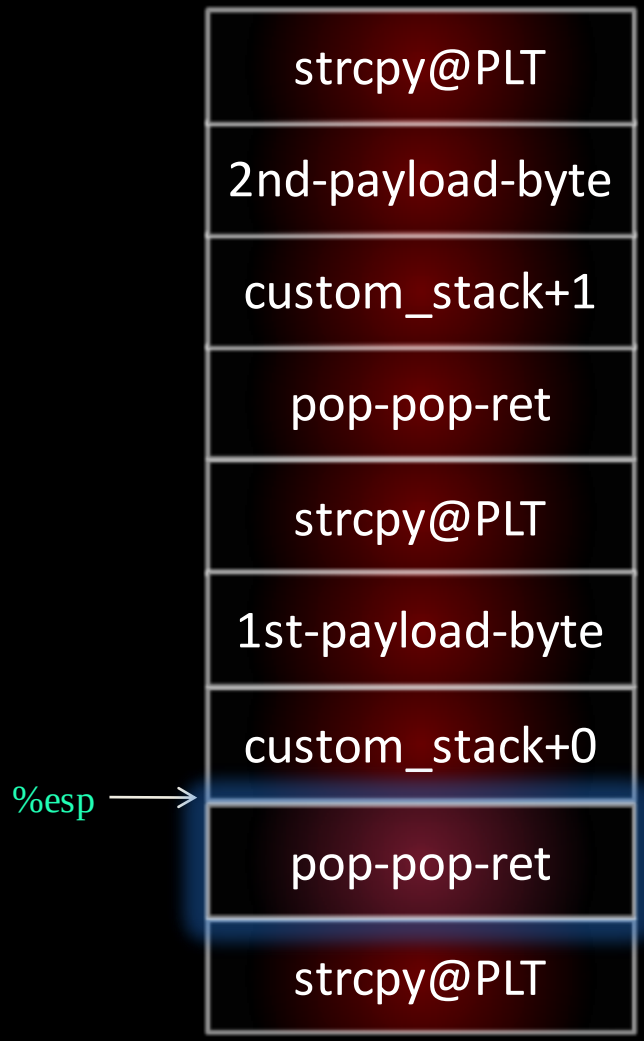


.text

```
<strcpy>:
. . . ; function prologue
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . . ; data copy instructions
ret
```

.data custom stack (fixed location)

Stage-0: Transfer payload to custom stack

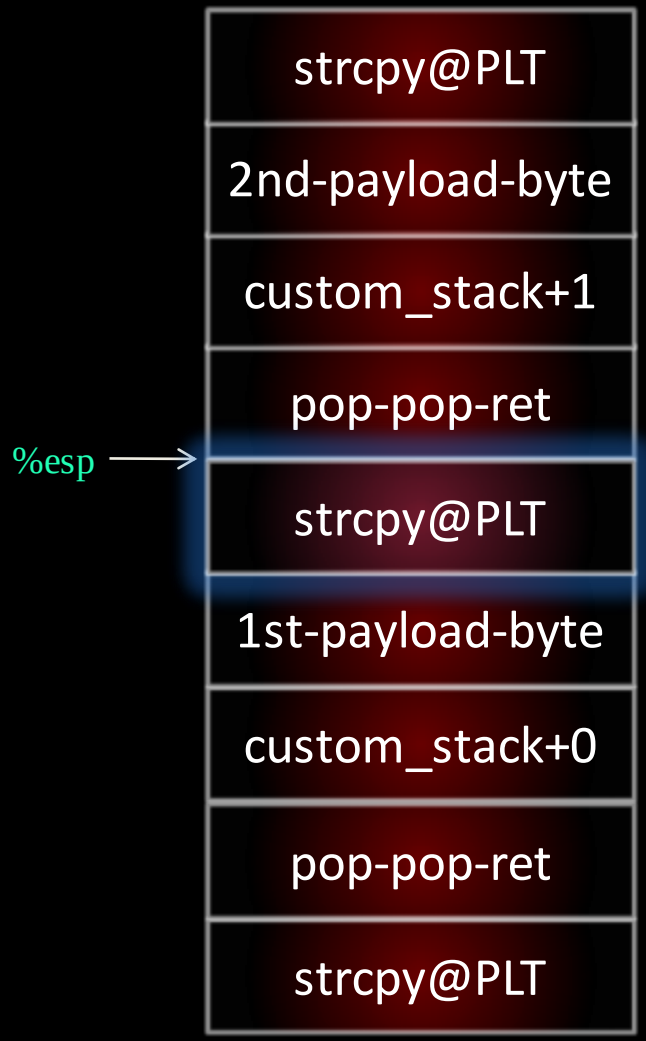


Payload = ['/bin/sh']

```
<strcpy>:
. . . ; function prologue
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . . ; data copy instructions
ret
```

.data custom stack (fixed location)

Stage-0: Transfer payload to custom stack

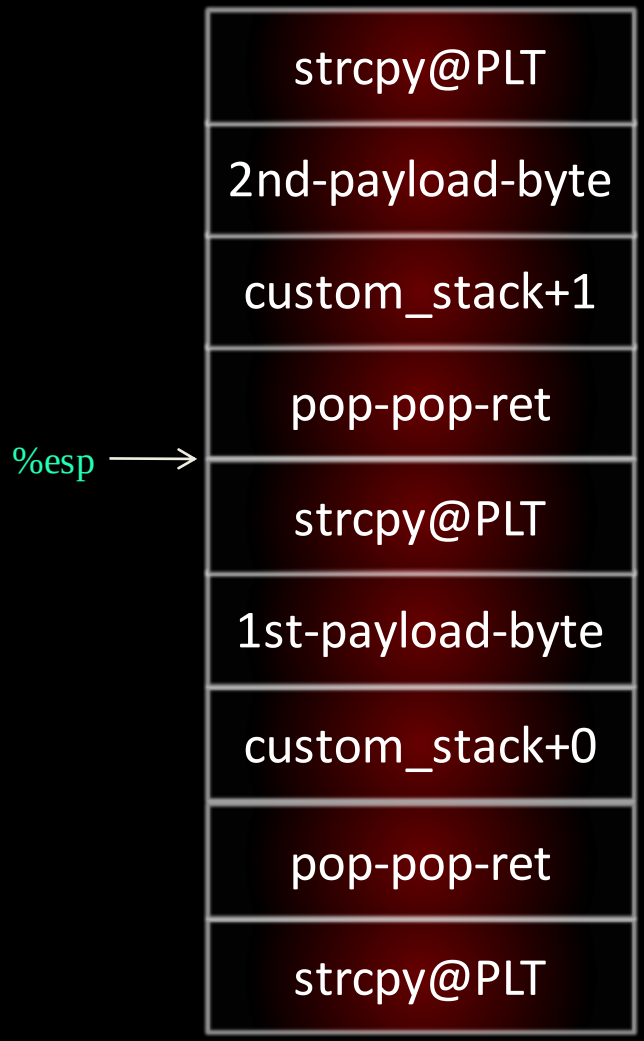


Payload = ['/bin/sh']

```
pop %ecx
pop %ebx
ret
```

.data custom stack (fixed location)

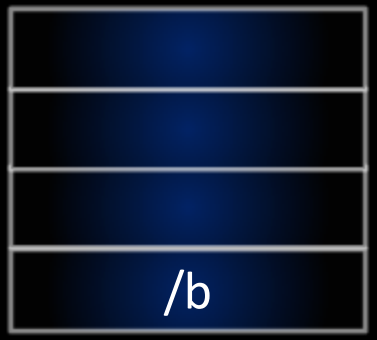
Stage-0: Transfer payload to custom stack



Payload = ['/bin/sh']

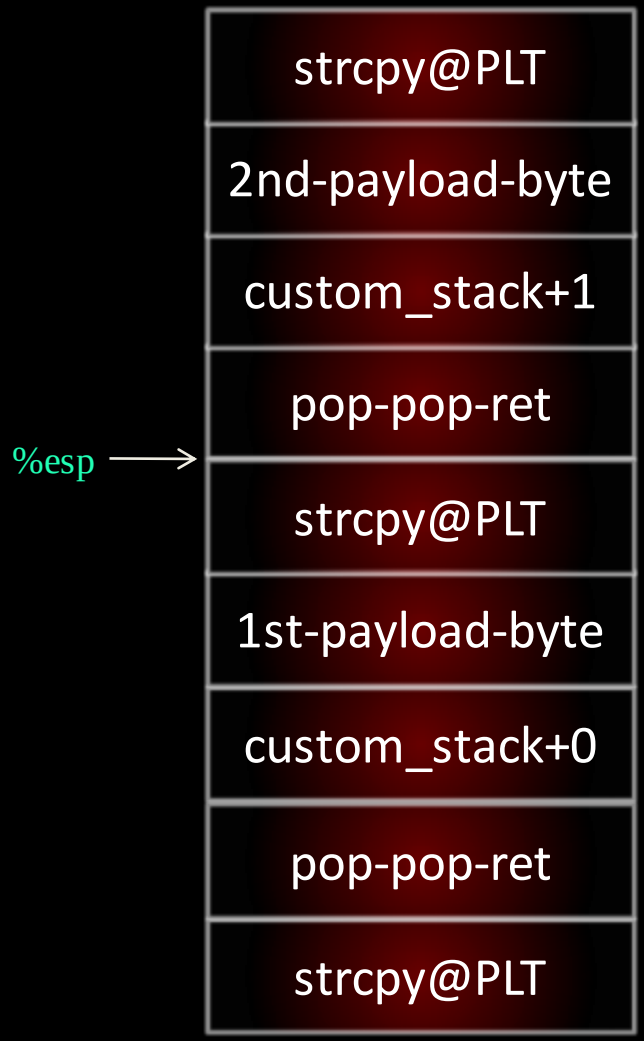
.text

```
<strcpy>:
. . . ; function prologue
mov     0x8(%ebp), %eax
mov     0xc(%ebp), %esi
. . . ; data copy instructions
ret
```



.data custom stack (fixed location)

Stage-0: Transfer payload to custom stack



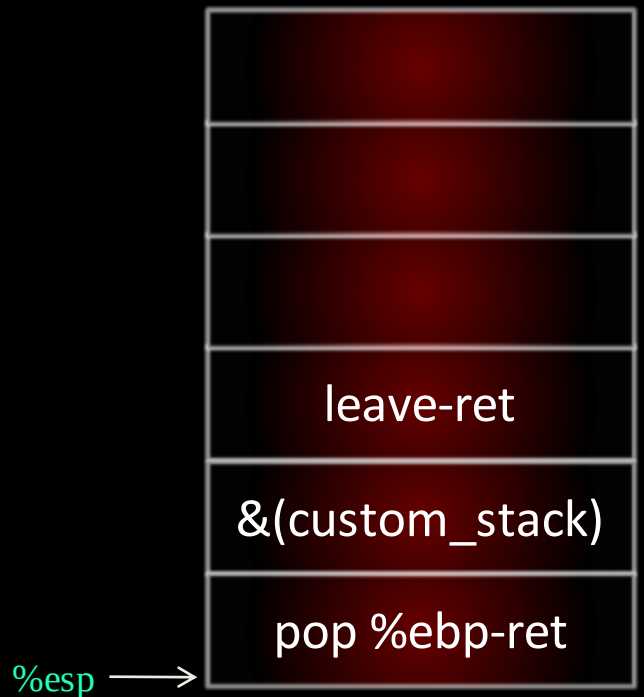
Payload = ['/bin/sh']

Some instructions later…

.data custom stack (fixed location)

From Stage-0 to Stage-1

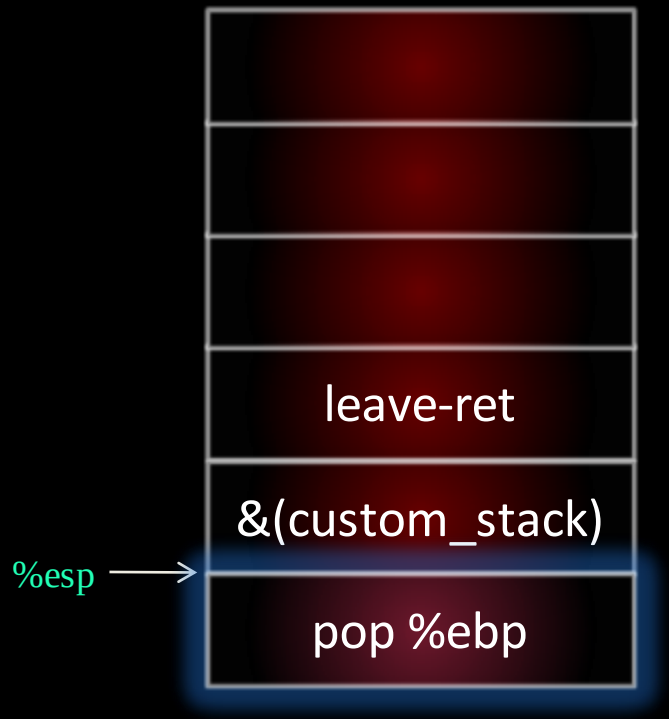
Switch stack pointer to custom stack



.data custom stack (fixed location)

From Stage-0 to Stage-1

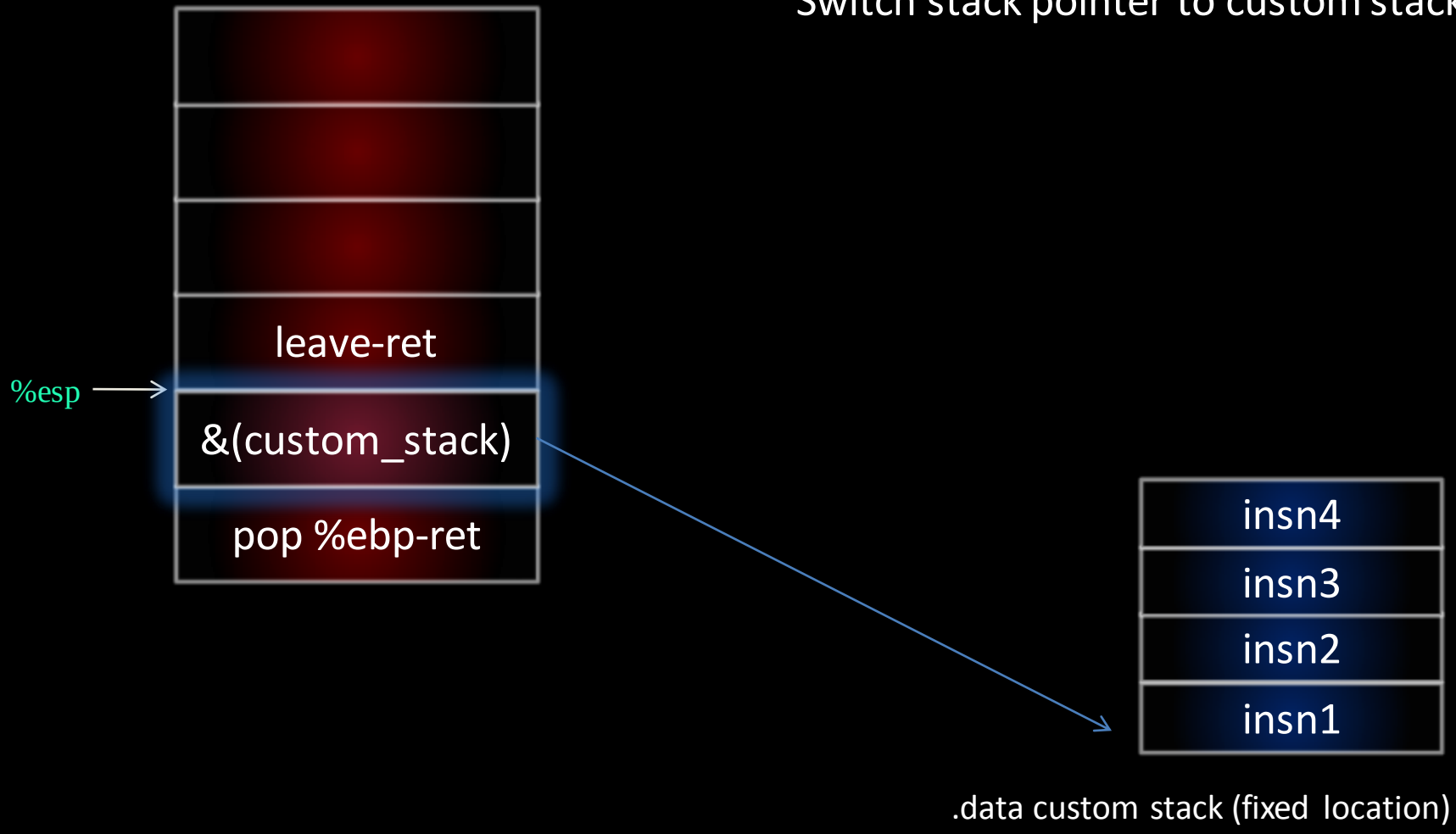
Switch stack pointer to custom stack



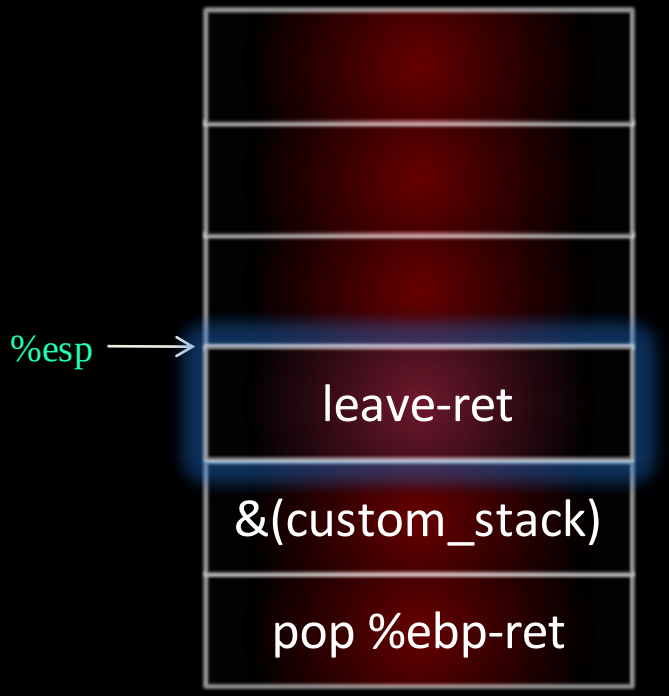
.data custom stack (fixed location)

From Stage-0 to Stage-1

Switch stack pointer to custom stack



From Stage-0 to Stage-1

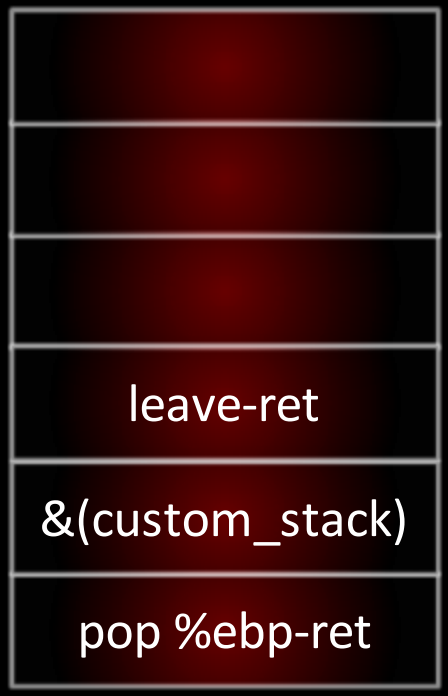


Switch stack pointer to custom stack

```
leave:
        movl %ebp, %esp
        pop %ebp
ret
```

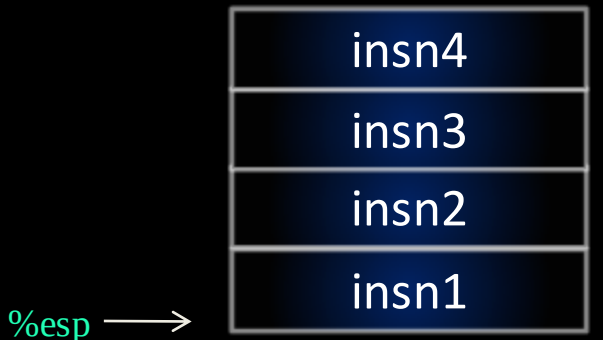
.data custom stack (fixed location)

From Stage-0 to Stage-1



Switch stack pointer to custom stack

```
leave:
        movl %ebp, %esp
        pop %ebp
ret
```



.data custom stack (fixed location)

Stage 1

- Possible payloads
  - `Chained ret-to-libc calls`
  - `Return to: mprotect() + memcpy() + mprotect`
  - `ROP Shellcode`

Stage 1

- Possible payloads
  - `Chained ret-to-libc calls`
  - `Return to: mprotect() + memcpy() + mprotect`
  - `ROP Shellcode`

DEMO

Other Affected Technologies

- Hardware protected machines
- Code Signing
  - Xbox
  - iPhone OS

Outro

- NX Memory is not enough
- **ASLR makes things harder (it may still be feasible to exploit)**
- **Secure coding is still important**

Thanks ☺

Q & A

Rubén Ventura Piña (tr3w)
tr3w@tr3w.net
http://tr3w.net
Twitter: trew_0

Shouts!:
Ayzax, hkm, KBrown, lightos, nitr0us, sirdarckcat