# Memory Corruption

Internet Security

Adrian Dabrowski, **Georg Merzdovnik**, Aljosha Judmayer,
Johanna Ullrich, Christian Kudera

# News from the Lab



- Challenge 4 ended
  - 145 solves!
- Challenge 5 starts today at 20:00

# News from the Lab - FaustCTF



- FaustCTF will be held on **Fri, 01 Jun. 2018, 15:00**
  - We will give an introduction for Attack/Defense CTFs (How? What? Why?)
  - Today, 17:30, @EI 3A
  - (Attendance is not Mandatory ;) )
- If you are interested:
  - Write a mail to `ctf@w0y.at`
  - And/or come to the meetup today

# News from the Field



- Spectre-NG
  - Spectre abuses speculative execution to access memory
  - Spectre-NG proposes some new attack vectors (2 released, 8 to come)
  - Problem in Architecture
- How bad is it really?
  - Patches currently introduce a lot overhead
  - latest Microcode patches introduce 2% to 8%
  - However, attacks are more likely to hit servers than end users (easier to exploit)

# Overview



1. Memory Corruption
2. The Stack
3. Taking Control of the Program
4. Protection and Prevention Mechanisms
(and how to circumvent them)
5. Return Oriented Programming (ROP)

# Memory Corruption

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations
- Result from mistakes done while writing code, because of
  - unfamiliarity with language
  - ignorance about security issues
  - unwillingness to take extra effort
- Vulnerable software
  - mostly C / C++ programs
  - not in languages with automatic memory management
    - dynamic bounds checks (e.g., Java)
    - automatic resizing of buffers (e.g., Perl)

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         "secret");

for (int i=0; 1; i++) {
  char b = getchar();
  if (b == '\n') break;
  data[i] = b;
}
```

password

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
```

data

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
```

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         "secret");

for (int i=0; 1; i++) {
  char b = getchar();
  if (b == '\n') break;
  data[i] = b;
}
```

password (snprintf)

```
00 00 00 00
00 00 00 00
"e  t \0"00
"s  e  c  r"
```

data

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
```

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         "secret");

for (int i=0; 1; i++) {
  char b = getchar();
  if (b == '\n') break;
  data[i] = b;
}
```

password

```
00 00 00 00
00 00 00 00
"e  t \0"00
"s  e  c  r"
```

data (getchar)

```
00 00 00 00
00 00 00 00
00 00 00 00
41 00 00 00
```

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         "secret");

for (int i=0; 1; i++) {
  char b = getchar();
  if (b == '\n') break;
  data[i] = b;
}
```

password

```
00 00 00 00
00 00 00 00
"e  t \0"00
"s  e  c  r"
```

data (getchar)

```
00 00 00 00
00 00 00 00
00 00 00 00
41 41 00 00
```

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         "secret");

for (int i=0; 1; i++) {
  char b = getchar();
  if (b == '\n') break;
  data[i] = b;
}
```

password

```
00 00 00 00
00 00 00 00
"e  t \0"00
"s  e  c  r"
```

data (getchar)

```
00 00 00 00
41 00 00 00
41 41 41 41
41 41 41 41
```

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         "secret");

for (int i=0; 1; i++) {
  char b = getchar();
  if (b == '\n') break;
  data[i] = b;
}
```

password

```
00 00 00 00
00 00 00 00
"e  t \0"00
41 41 "c  r"
```

data (getchar)

```
41 41 41 41
41 41 41 41
41 41 41 41
41 41 41 41
```

# Buffer Overflows



- A buffer overflow occurs any time a program attempts to store data beyond the boundaries of a buffer, overwriting the subsequent memory locations

```
char password[16];
char data[16];

snprintf(password,
         sizeof(password)-1,
         "secret");

for (int i=0; 1; i++) {
  char b = getchar();
  if (b == '\n') break;
  data[i] = b;
}
```

password

```
41 41 41 41
41 41 41 41
41 41 41 41
41 41 41 41
```

data (getchar)

```
41 41 41 41
41 41 41 41
41 41 41 41
41 41 41 41
```

# Buffer Overflow



- Goals
  - overwrite other “interesting” variables / memory locations (file names, passwords, pointers…)
  - force the program to execute operations it was not intended to do:
    1. inject into (or simply find) code in the process memory
    2. change flow of control (flow of execution) to execute that code
- Common targets
  - Setuid/Setgid programs: elevate privileges
  - Network servers: remote access
  - Client Programs (e.g Browsers, Office,…)

# Buffer Overflow - History



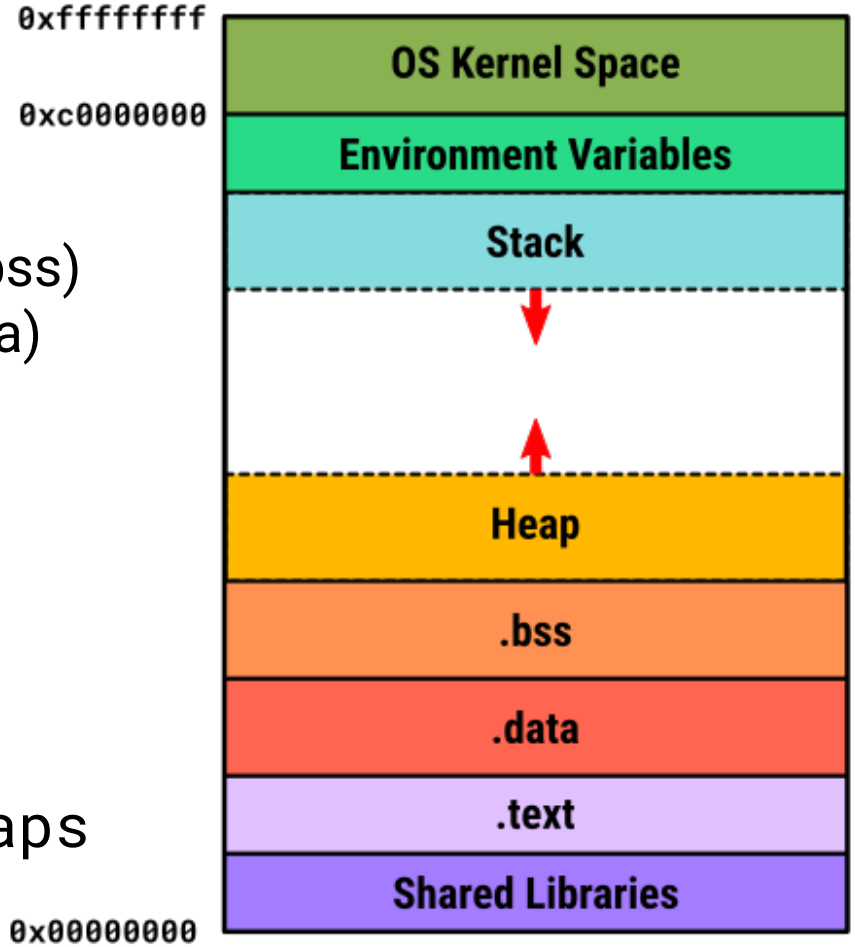
- Morris worm (1988): overflow in “fingerd”
  - 6,000 machines infected (10% of the Internet)
- SQL Slammer (2003): overflow in MS-SQL server
  - 75,000 machines infected in 10 minutes
- In 2003, around 75% of the vulnerabilities were buffer overflow (CERT)
- In 2012, around 35% buffer overflows, massive gain of web vulnerabilities
- Today, web application attacks are becoming more common, but for servers/clients using outside input, BO remain important
- Embedded Systems are usually also not as well protected…

# The Stack

# Memory Layout



- Stack segment
  - local variables
  - procedure activation records (i.e. return address, function parameters, etc.)
- Data segment
  - global uninitialized variables (.bss)
  - global initialized variables (.data)
  - dynamic variables (heap)
- Code (text) segment
  - program instructions
  - usually read-only
- Linux: `$ cat /proc/<pid>/maps`



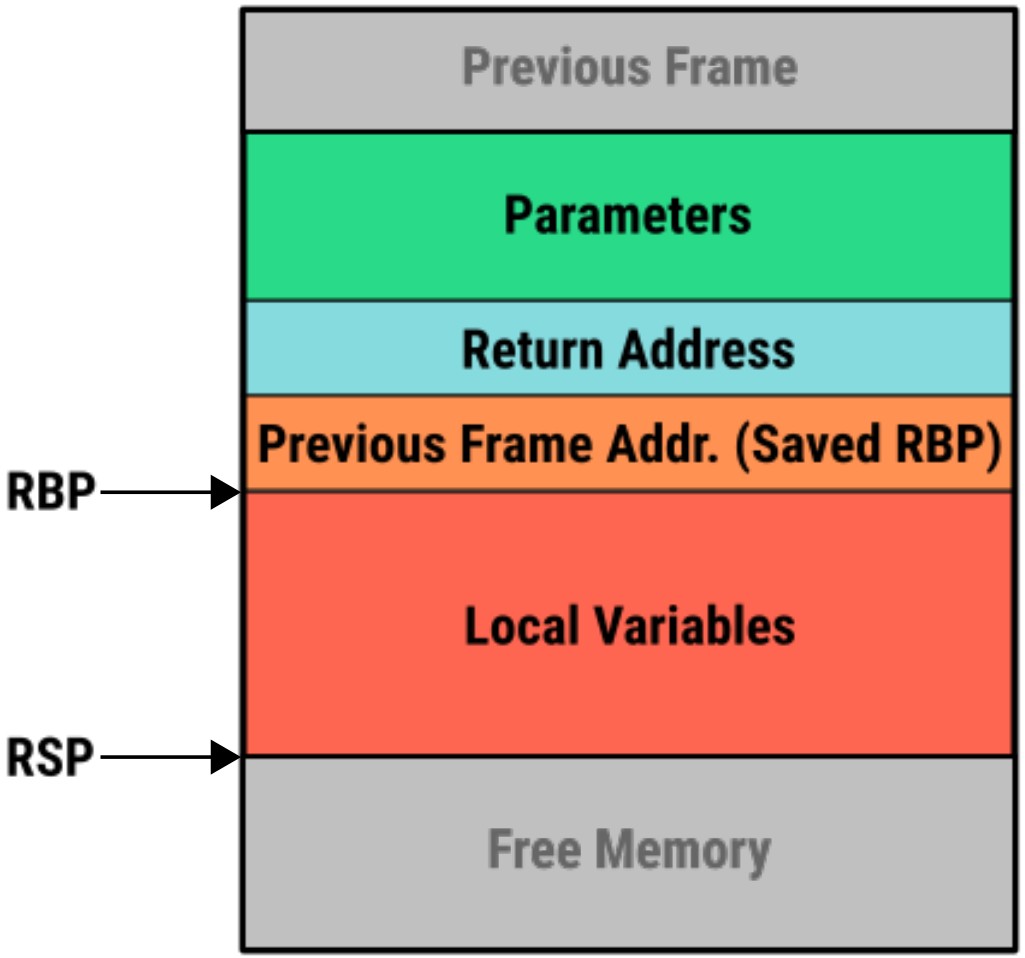
# Stack



- Usually grows towards smaller memory addresses
  - Intel, Motorola, SPARC, MIPS
- Special processor register points to top of stack
  - *stack pointer - SP*
  - points to last stack element
- Composed of frames
  - upon function call, a new frame is pushed on top of stack
  - used to conveniently reference local variables
  - upon function return, frame is discarded, last frame on stack is restored
  - address of current frame stored in processor register
    - *frame/base pointer - BP*

# Frame




Previous Frame
Parameters
Return Address
Previous Frame Addr. (Saved RBP)
RBP
Local Variables
RSP
Free Memory

# Taking Control of the Program

# The Idea



- Overwrite a pointer with the address of our code
- First: locate a pointer that (eventually) will be copied to the EIP register or that points to data that will be copied to the EIP
  - function pointers (on the stack, heap, BSS…)
  - saved EBP
  - procedure return address
  - entry in the GOT
  - jmp_buf
- Second: overwrite the pointer with our value
  - stack overflow
  - heap overflow
  - format string

# Smashing the Stack



- A procedure contains a local buffer variable allocated on the stack
- The procedure copies user-controlled data to the buffer without verifying that the data size is smaller than the buffer
  - many libc function are potentially dangerous: *strcpy, strcat, gets, fgets, sprintf, scanf, …*
- The user data overwrites the other variables on the stack, up to the return address saved in the function frame
- When the procedure returns, the program fetches the return address from the stack and copies it to the RIP register
- Since we can control the return address, we can execute our own code!

# Finding the right offset



- Usually easy to segfault the program
- But how to know the exact offset in your buffer where your return-address needs to be written
- Use a *De Bruijn Sequence*

# De Bruijn Sequence



- cyclic sequence in which every possible substring of length *n* occurs exactly once
- How can we use this?
  - generate a de-brujin sequence
  - Write it into the buffer
  - Look where it segfaults (where it wanted to jump)
  - Look up this address in the sequence
  - Know the offset
- There is tool support for that (e.g. in Metasploit framework)

# Simple Example



- Sequence:
  - aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama
- Segfault at `0x61666161`
- This is the string sequence **aafa**
  - Beware byte order
- Lookup will yield offset 18
  - aaaabaaacaaadaaaea**aafa**aagaaahaaaiaaajaaakaaalaaama
  - So you need to write 18 bytes before your return address

# Choosing Where to Jump



- Address inside a buffer of which the attacker controls the content
  - PRO: works for remote attacks
  - CON: the attacker needs to know the address of the buffer, the memory page containing the buffer must be executable
- Address of a environment variable
  - PRO: easy to implement, works with tiny buffers
  - CON: only for local exploits, some programs clean the environment, the stack must be executable
- Address of a function inside the program
  - PRO: works for remote attacks, does not require an executable stack
  - CON: need to find the right code, one or more fake frames must be put on the stack

# Jumping into the Buffer



- The buffer that we are overflowing is usually a good place to put the code (shellcode) that we want to execute
- The buffer is somewhere on the stack, but in most cases the exact address is unknown
  - the address must be precise: jumping one byte before or after would just make the application crash
  - on the local system it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine
  - any change to the environment variables affect the stack position

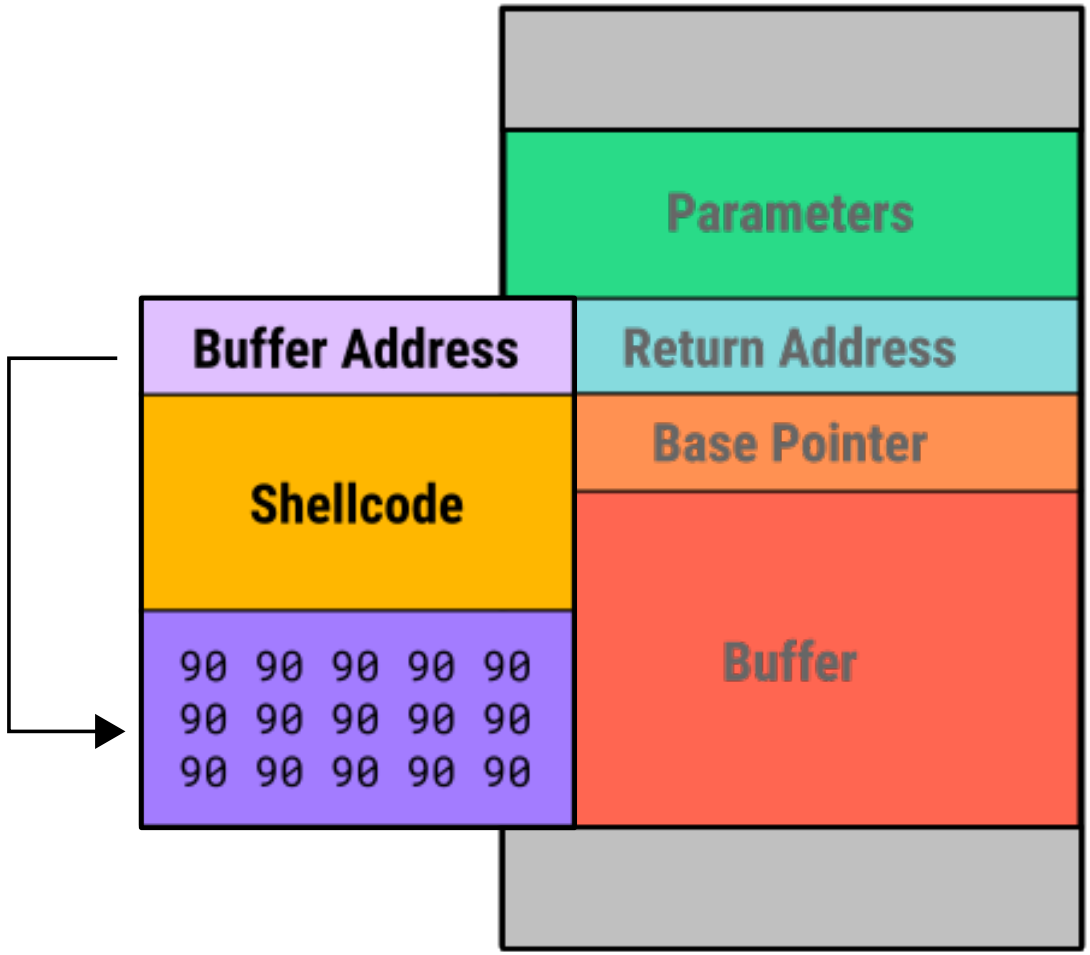
# Solution 1: The NOP Sled



- A sled is a “landing area” that is put in front of the shellcode
- Must be created in a way such that wherever the program jumps into it (independently of the offset)..
  - .. it always finds a valid instruction
  - .. it always reaches the end of the sled and the beginning of the shellcode
- The simplest sled is a sequence of no operation (NOP) instructions
  - Single byte instruction (0x90) that does not do anything
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area
  - Can make Bruteforce feasibly (e.g. increment by size of NOPSled instead of 1)

# Assembling the Malicious buffer





# Solution 2: Jump using a Register



- Find a register that points to the buffer (or somewhere into it)
  - ESP
  - EAX (return value of a function call)
- Locate an instruction that jumps/calls using that register
  - can also be in one of the libraries
  - does not even need to be a real instruction, just look for the right sequence of bytes, e.g.

```
<code>: ff e4    jmp *esp
```

- Overwrite the return address with the address of that instruction

# Solution 3: Heap Spraying



- We will come back to that in **Advanced Internet Security**

# Shellcode



- Sequence of machine instructions that is executed when the attack is successful
- Traditionally, the goal was to spawn a shell (that explains the name `shell code`)
- They can do practically anything:
  - create a new user
  - change a user password
  - modify the .rhost file
  - bind a shell to a port (remote shell)
  - open a connection to the attacker machine (reverse shell)

# How to spawn a Shell



```
int execve(char *file, char *argv[], char *env[])
```

- Three parameters:
  - ***file**: put the zero-terminated string `/bin/sh` somewhere in memory
  - ***argv[]**: put the address of the string `/bin/sh` followed by NULL (0x00000000) somewhere in memory
  - ***env[]**: put a NULL somewhere in memory

# The Address Problem (…on 32 bit)

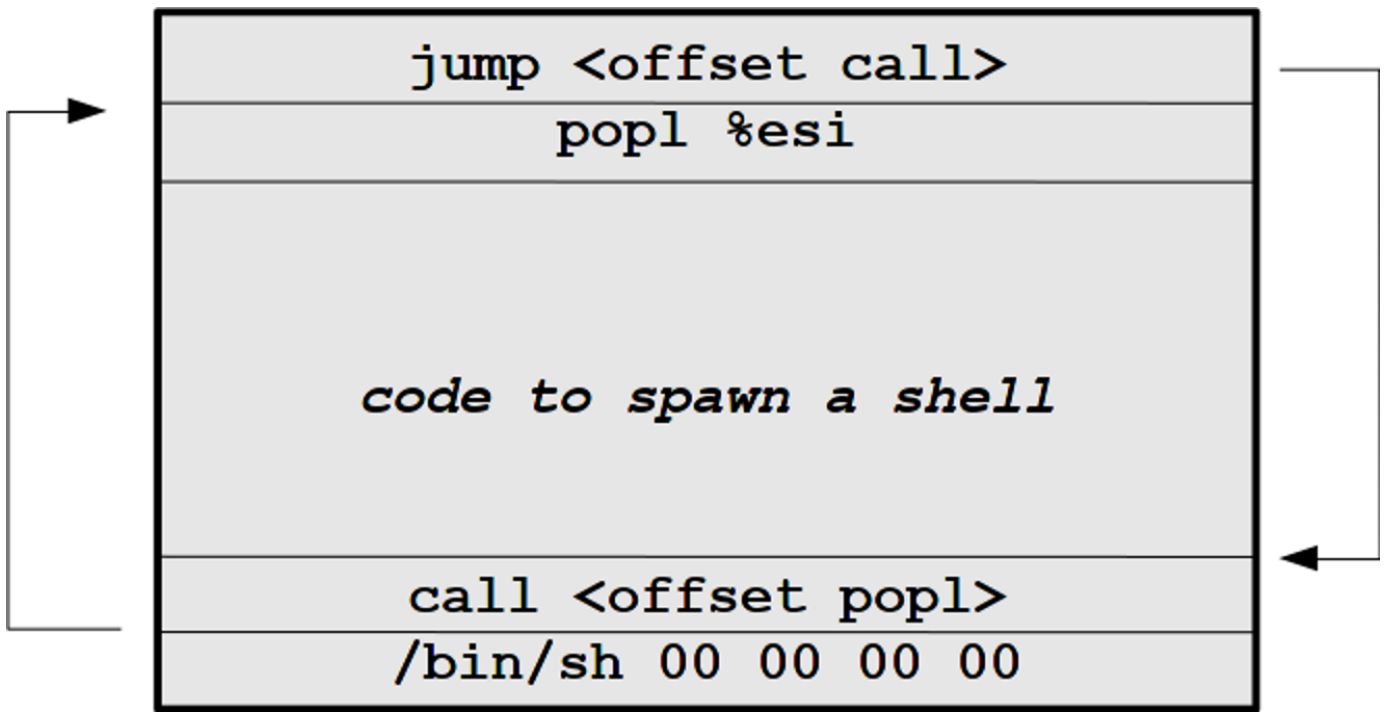


- How can we put in memory the address of the string `/bin/sh` if we do not even know where the position of the shellcode is?
- Solution…
  - the **CALL** instruction puts the return address on the stack
  - if we put a **CALL** instruction just before the string `/bin/sh`, when it is executed it will push the address of the string onto the stack
- (Actually there are more solutions, like using floating point instructions)

# The jump/call trick



- *popl* gets the return address set by the call instruction from the stack (that is, the address of /bin/sh)



# The Address Problem (…on 64 bit)



- Easier to get current address of shellcode
  - We can always do RIP relative addressing now
- Or use the 32bit techniques

# The Zeroes Problem



```
char shellcode[] =
    "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00"
    "\xc7\x46\x0c\x00\x00\x00\x00\xb8\x0b\x00"
    "\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
    "\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00"
    "\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f"
    "\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d"
    "\xc3";
```

- The shellcode is usually copied into a string buffer
- `\x00` is the string terminator character
- Problem: any null byte would stop copying

# The Zeroes Solution



- Solution: substitute any instruction containing zeros, with an alternative instruction

```
mov 0x0, reg -->
            xor reg, reg
mov 0x1, reg -->
            xor reg, reg
            inc reg
```

- Some tools provide this functionality automatically:
  - e.g., msfencode (metasploit framework)
  - alternative to shellcode modification: staging
    - encode shellcode (e.g., base64, eliminate unwanted chars)
    - decode before jumping to original code
  - might be helpful for a later challenge ;-)

# Protection and Prevention Mechanisms

## (and how to circumvent them)

# Exploit Mitigation

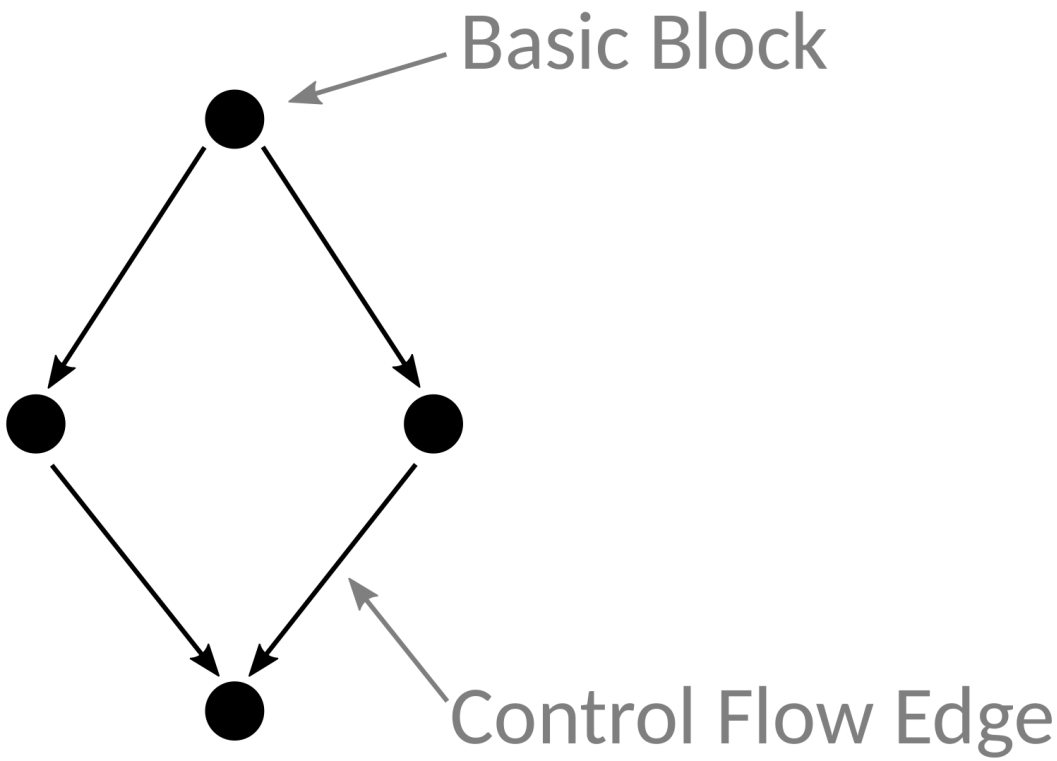


- Mitigate (prevent) system exploitation
  - Usually the goal is to prevent attackers from executing arbitrary code
- Several protections got invented to defend against attacks

# Mitigate What? - Arbitrary Code Execution



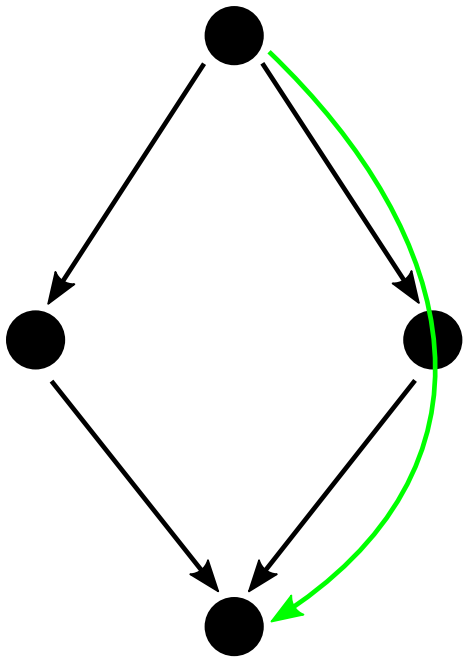
- Sample Control Flow Graph



# Mitigate What? - Arbitrary Code Execution



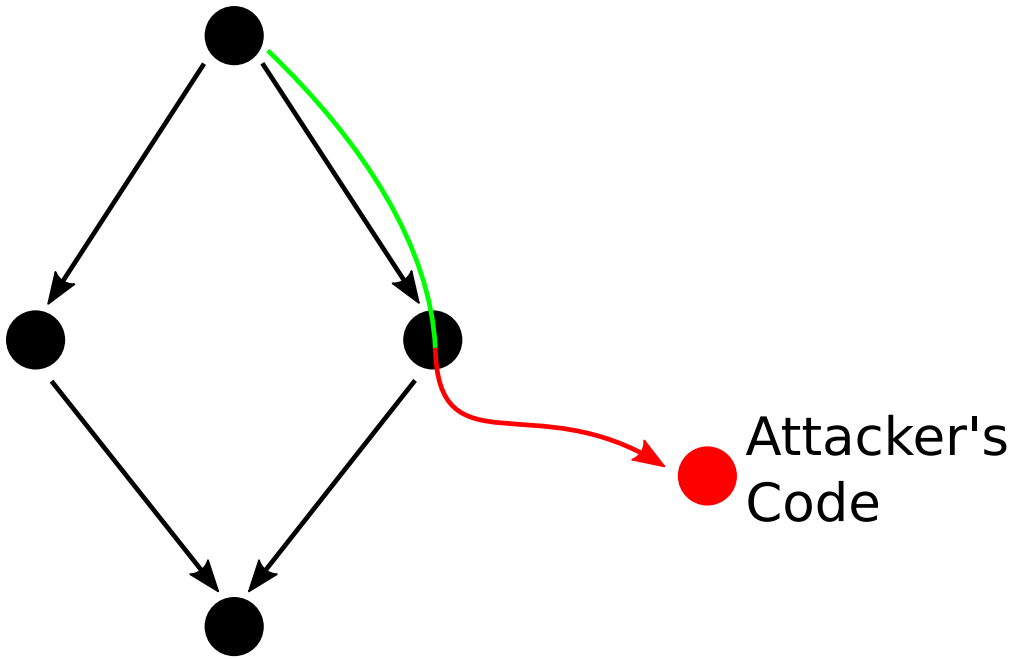
- Intended code path



# Mitigate What? - Arbitrary Code Execution



- Malicious path



# A Combination of Different Approaches



- At the program level
  - to prevent attacks by removing the vulnerabilities
- At the compiler level
  - to detect and block exploit attempts
- At the operating system level
  - to make the exploitation much more difficult

# Checksec.sh



- A simple bash script to check for enabled security features in binaries
- Checks performed:
  - RELRO
    - Relocation Read only (e.g. are sections like .GOT or .PLT writeable?)
  - Stack Canary
  - NX bit (DEP)
  - PIE (Position independent Executable)
  - RPATH/RUNPATH
    - Can set library search path overruling the normal system loader
  - FORTIFY (FORTIFY_SOURCE enabled?)
    - Adds buffer overflow checks to certain functions

```
┌─[atlas][/tmp]
└─▪ ./checksec.sh --file /bin/ls
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      FORTIFY Fortified Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE        No RPATH   No RUNPATH   Yes     5                 15      /bin/ls
```

http://www.trapkit.de/tools/checksec.html

# First of All: the Human Factor

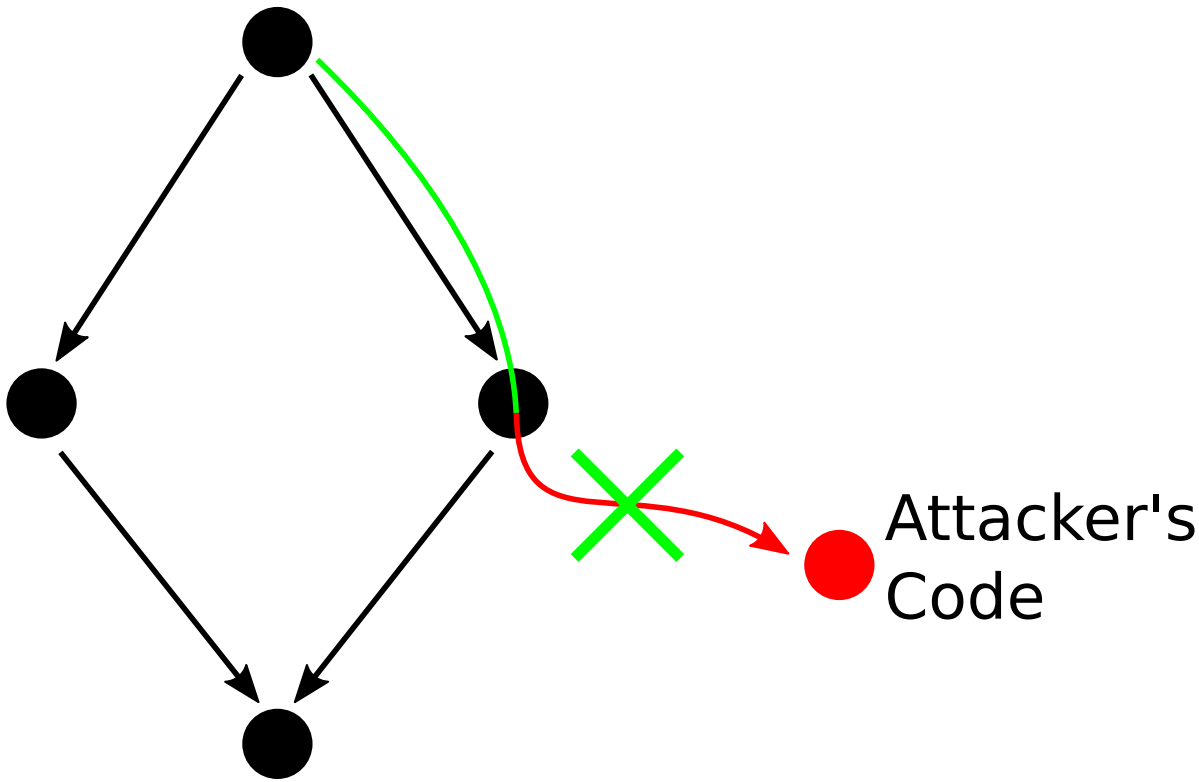


- The main cause of buffer overflows are stressed/bad programmers, not the C language ;-)
  - educate programmers how to write secure code
  - test the programs with a focus on security issues
- Switch to more secure library functions
  - Standard Library: strncpy, strncat, …
  - BDS's strlcpy, strlcat (boundary safe)
  - LibSafe: wrapper around a set of potentially "dangerous" libc functions
  - ContraPolice: libc extension to prevent heap overflow

# Protections - Canaries



- Prevent control transfer



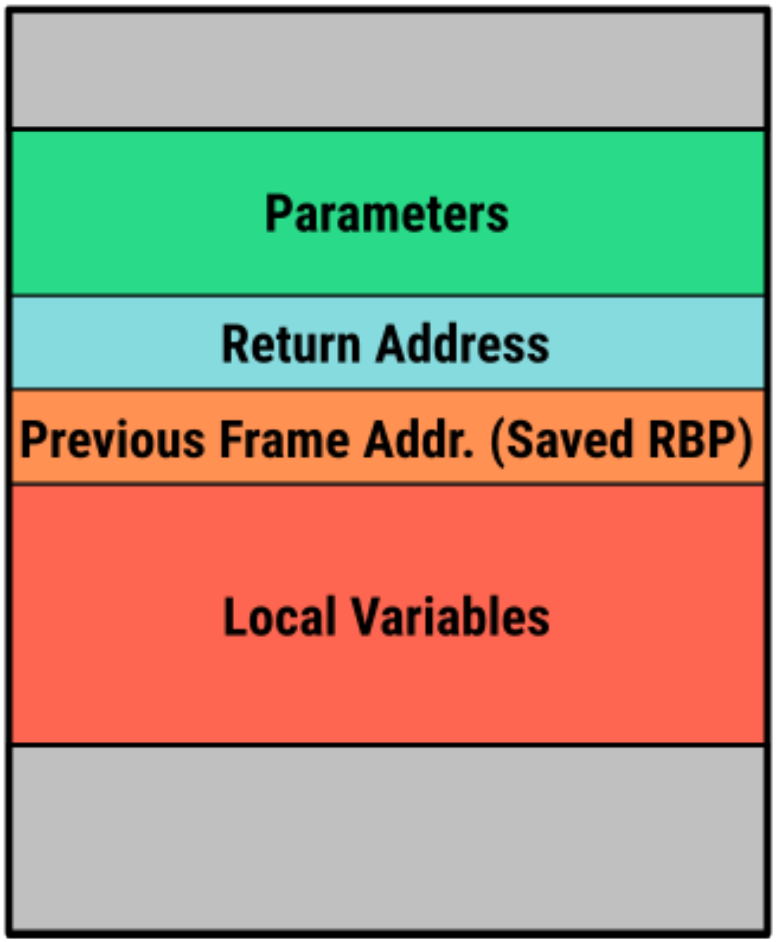
# Compile-time Technique: Stack Protection

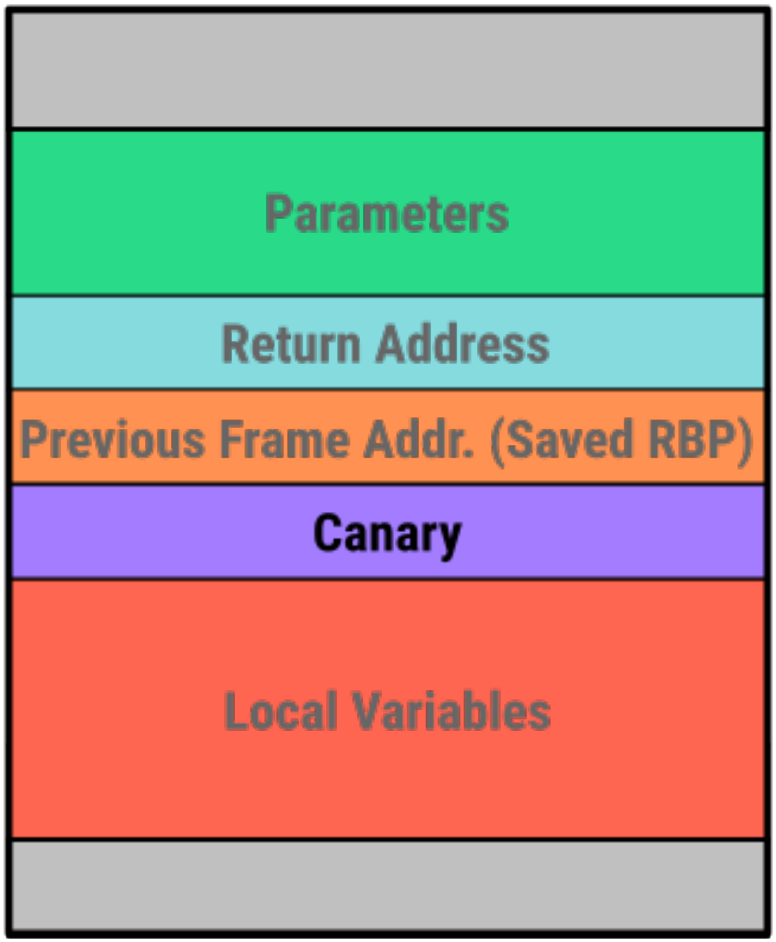


- Goal:
  - protect the function frame from being overwritten by the attacker
- Idea:
  - add a “canary” value between the local variables and the saved EBP
  - at the end of the function, check that the canary is “still alive”
  - a different canary value means that a buffer preceding it in memory has been overflowed

# Stack canary





# Canary Values



- **Terminator canaries:** contain string terminator characters (`\0`) to stop string copy routines
- **Random canaries:** contain a random value generated at program initialization and stored in a global variable the attacker has to find a way to read the canary
- **Random XOR canaries:** contain a random value XORed with all (or part of) the control data to protect can be used to detect attacks in which the attacker is able to modify the return address without overwriting the canary

# Stack Protection Implementations



- StackGuard
  - first canary implementation (by Immunix Corp) in 1997
  - implemented as a patch for gcc 2.95
- GCC Stack-Smashing Protector (ProPolice)
  - first developed as a patch for gcc 3.x
  - supports canary and stack variable rearrangement
  - part of GCC 4.1
- Visual Studio 2003 - GS option
  - compiler option to insert canaries (called security cookies by Microsoft), stack rearrangement

# Canary Circumvention

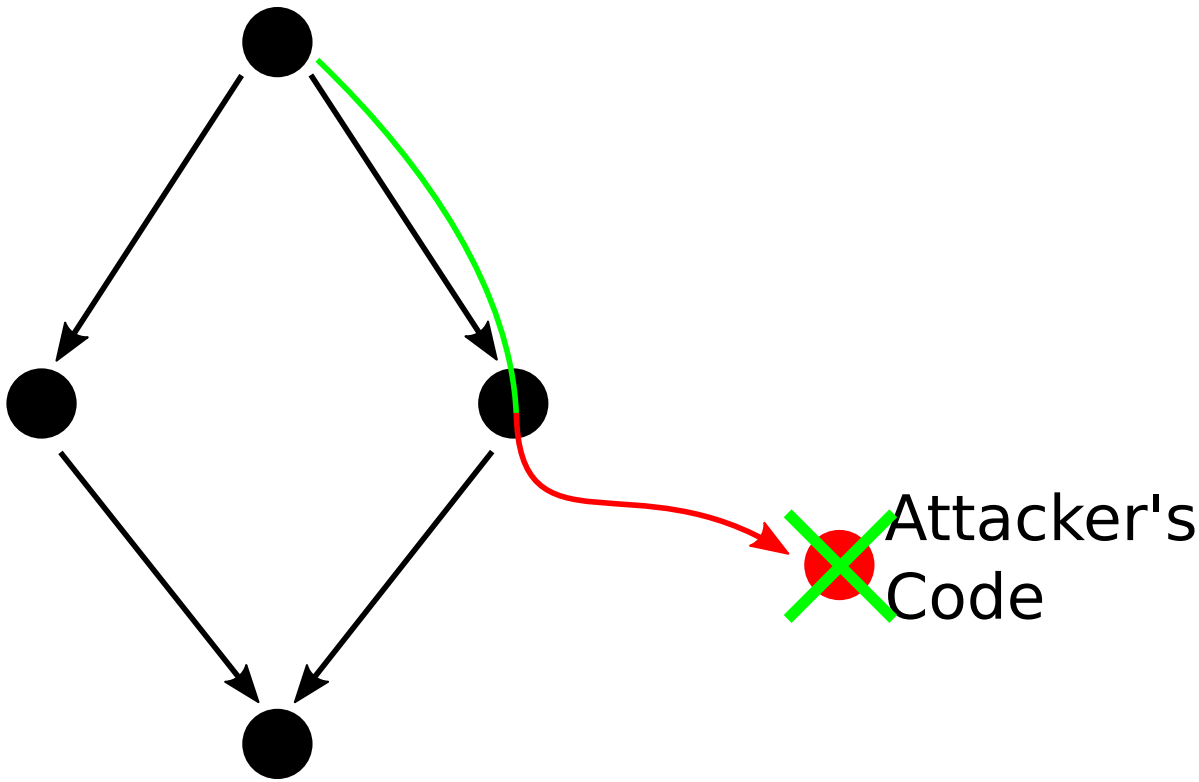


- **Terminator canaries:**
  - If the overflow is not based on string operations, just write Null bytes at the correct position and the canary will be fixed
  - Easy, since we know exactly what the canary looks like
- **Random canaries:**
  - Canaries are usually initialized at program startup
  - Usually they are fixed an not reset during execution
  - In a forking server they can be bruteforced
    - (maybe even byte by byte, just do not overwrite the whole canary at once. ;) )

# Protections - NX/DEP



- Prevent Execution



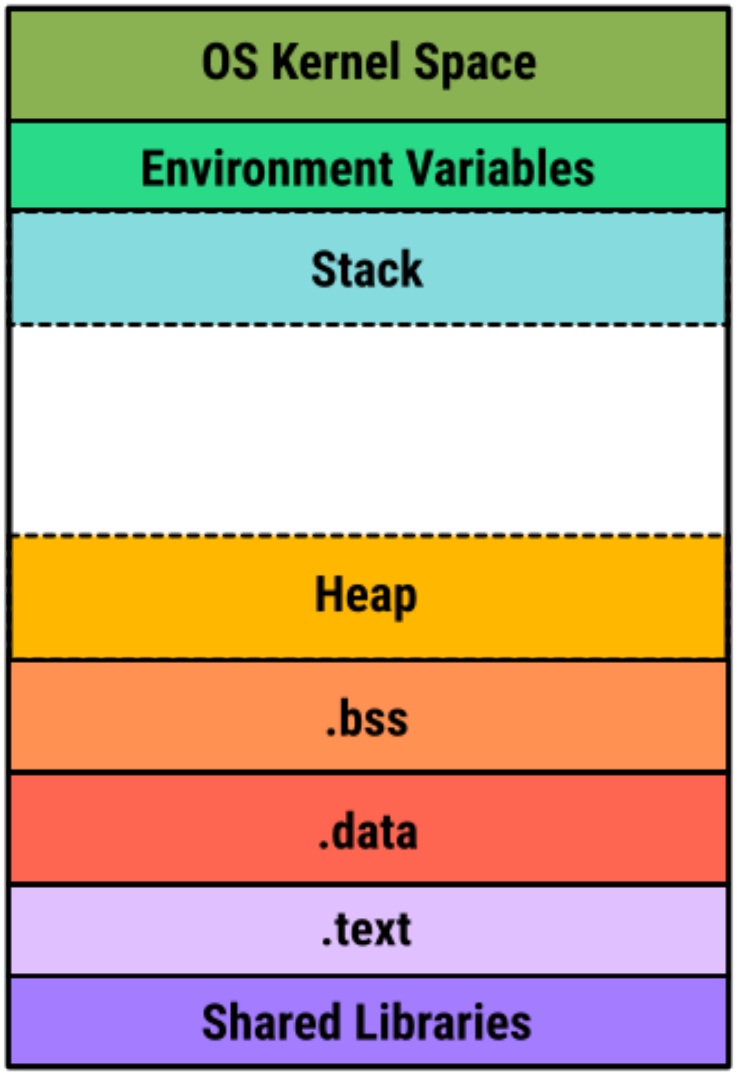
# Data Execution Prevention (DEP)



- Operating System/Run time based mitigation
- Ensures that only code segments are marked as executable
- Defense against shellcode payloads
  - Should prevent it’s execution
- Different names:
  - DEP, NX, W^X,…

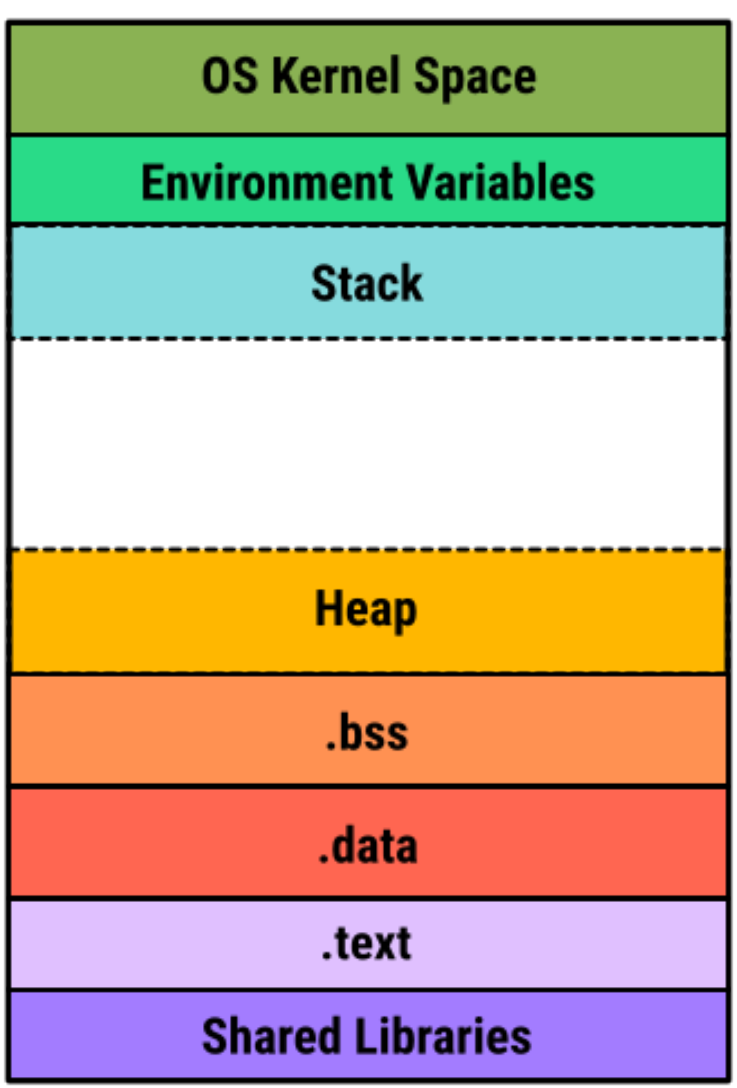
# Without DEP




0xffffffff
OS Kernel Space
Environment Variables
Stack
RWX (Read, Write, Execute)
Heap
RWX (Read, Write, Execute)
.bss
RW- (Read, Write)
.data
RW- (Read, Write)
.text
R-X (Read, Execute)
Shared Libraries
0x00000000

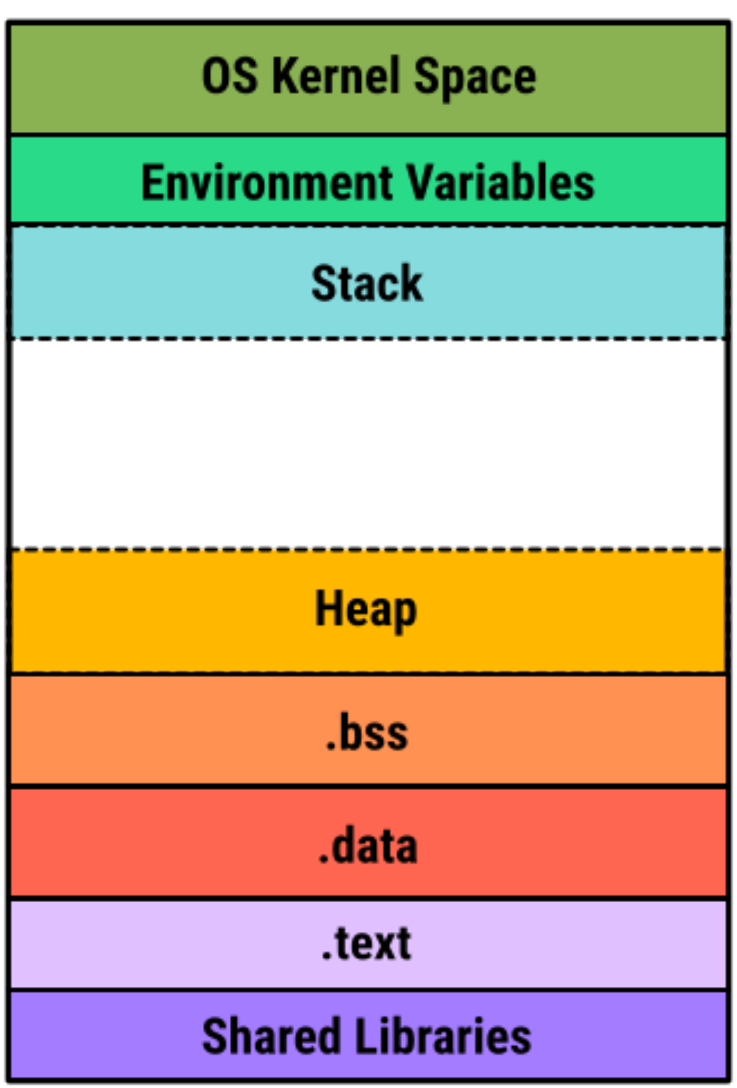
# Without DEP




0xffffffff
OS Kernel Space
Environment Variables
Stack
RWX (Read, Write, Execute)
Heap
RWX (Read, Write, Execute)
.bss
RW- (Read, Write)
.data
RW- (Read, Write)
.text
R-X (Read, Execute)
Shared Libraries
0x00000000

# Without DEP




0xffffffff
OS Kernel Space
Environment Variables
Stack
RW- (Read, Write, Execute)
Heap
RW- (Read, Write, Execute)
.bss
RW- (Read, Write)
.data
RW- (Read, Write)
.text
R-X (Read, Execute)
Shared Libraries
0x00000000

# DEP

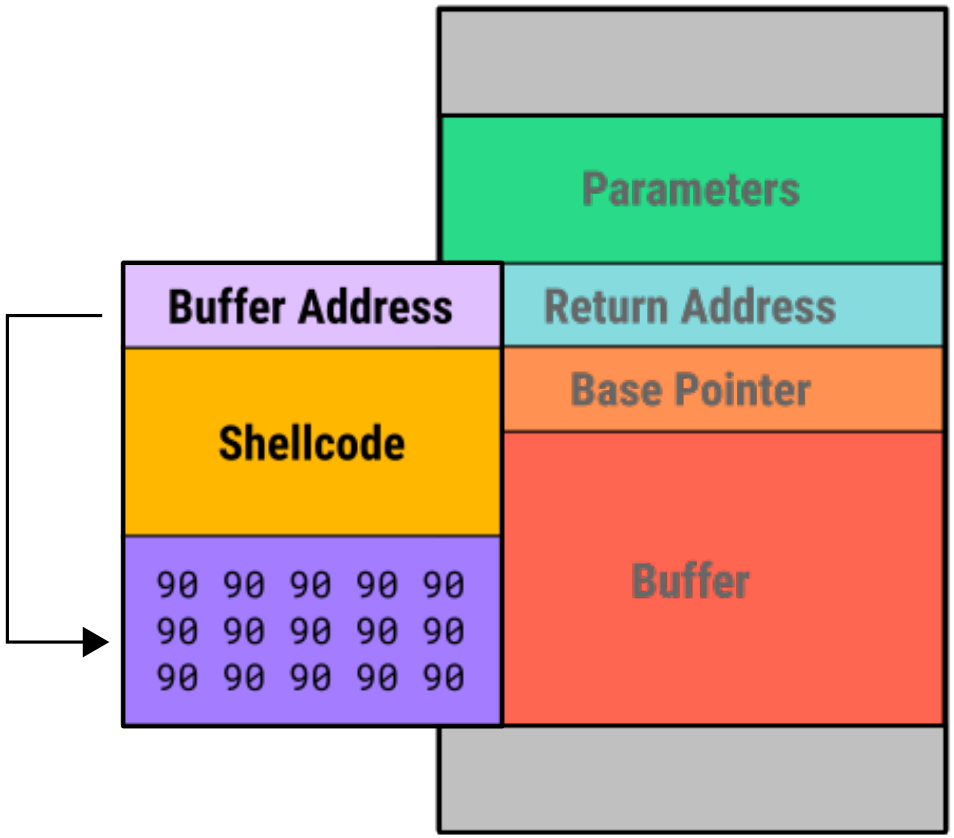


- No segment of memory should ever be Writable and Executable at the same time, ‘W^X’
- Common data segments
  - Stack, Heap
  - .bss
  - .ro
  - .data
- Common code segments
  - .text
  - .plt

# What about our shellcode?



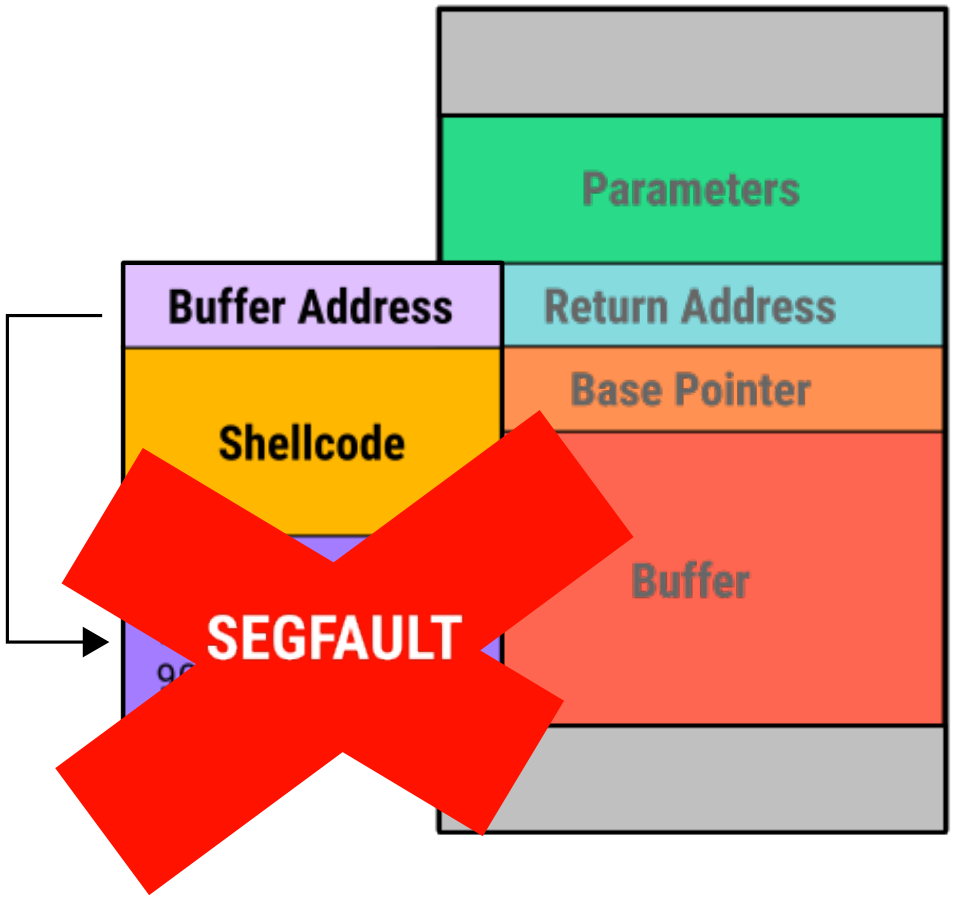
- Without DEP



# What about our shellcode?



- With DEP



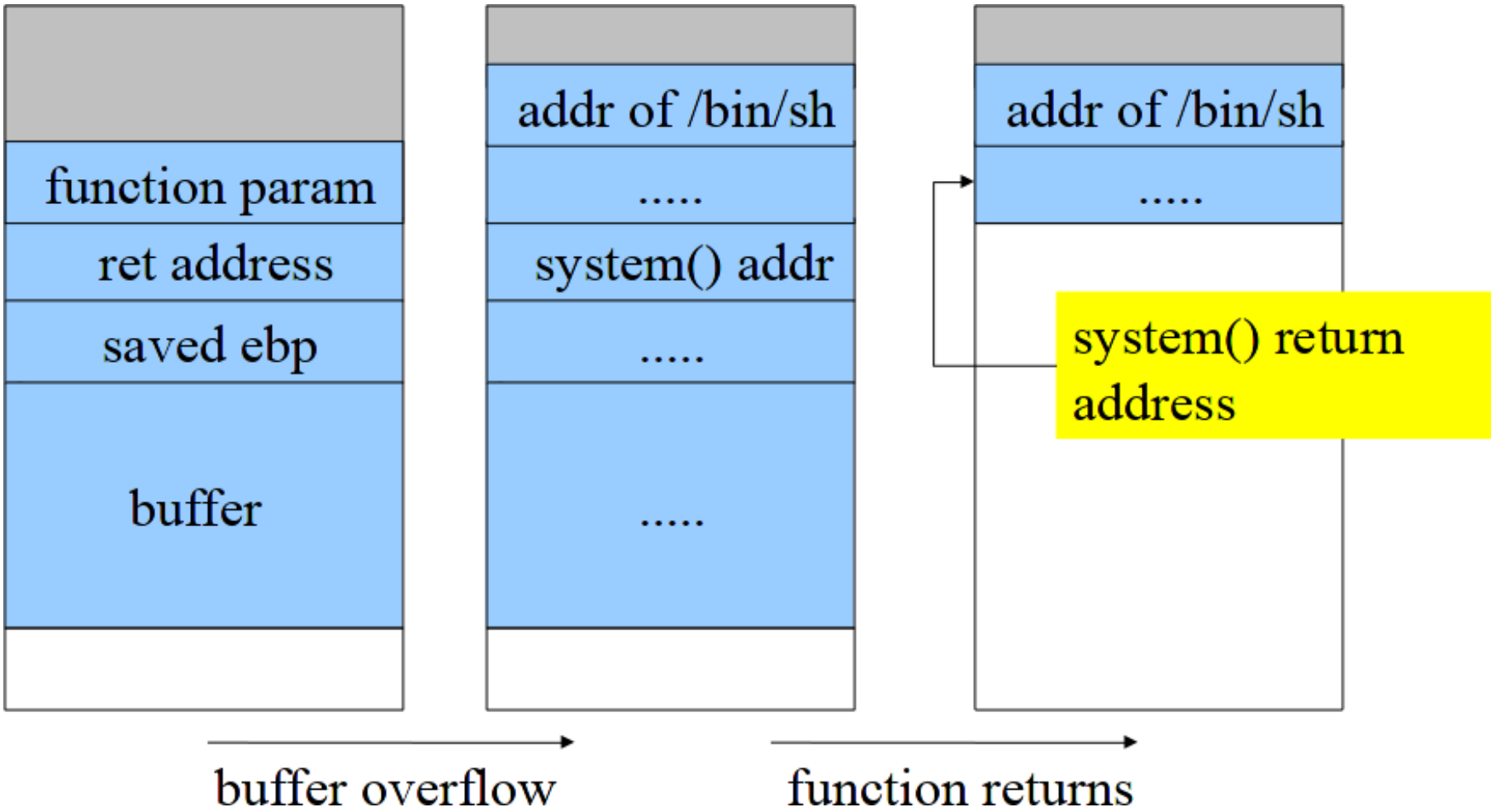
# Bypassing DEP



- The shellcode in the buffer cannot be executed but..
  - the attacker can still control the stack content
  - the attacker can still control the EIP value
- Why not call existing code?
  - This is usually some form of Return Oriented Programming
- libc is an attractive target
  - very powerful functions (`system()`, `execve()`, …)
  - linked by almost every program

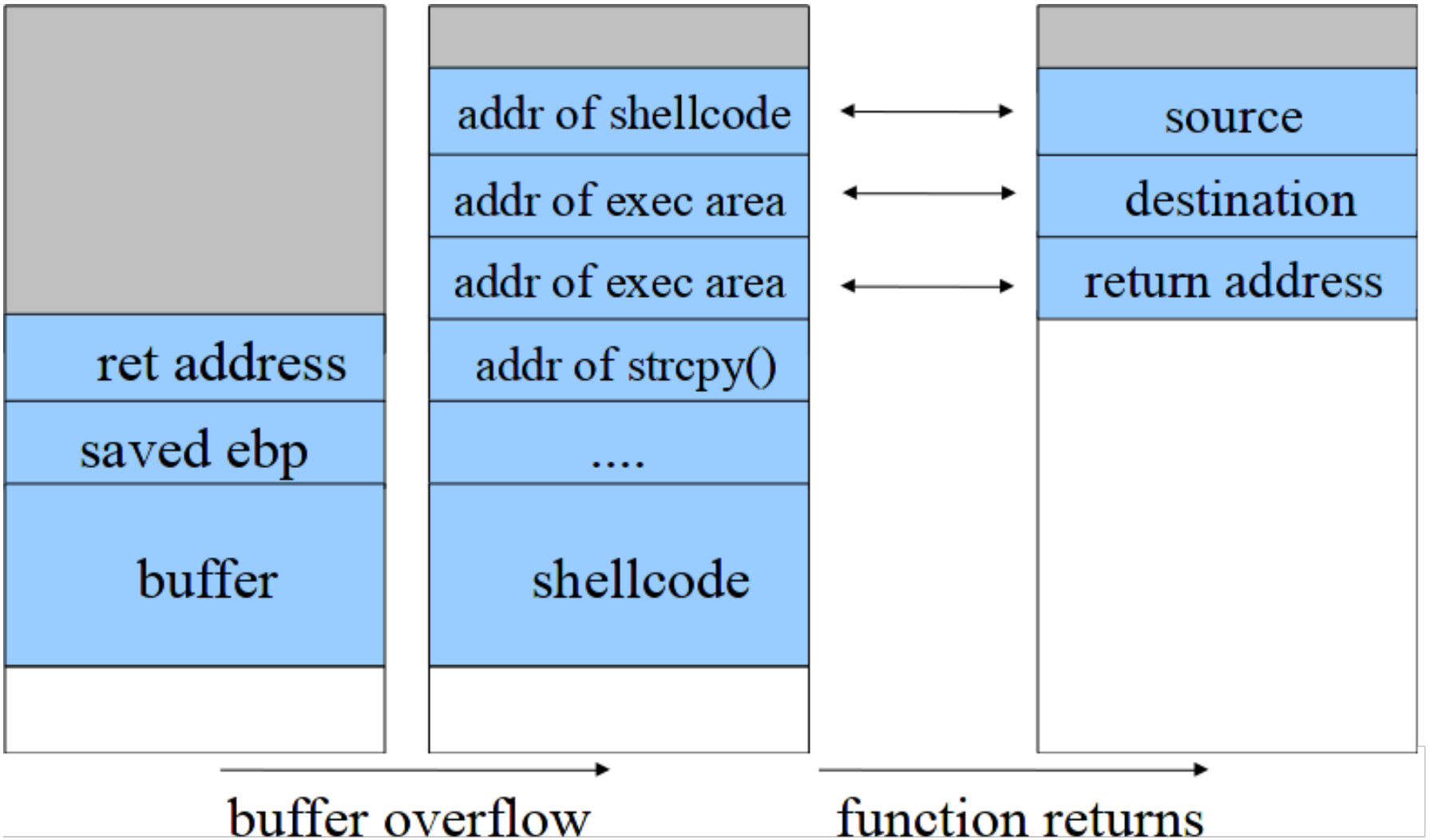
# Return into libc (return2libc)





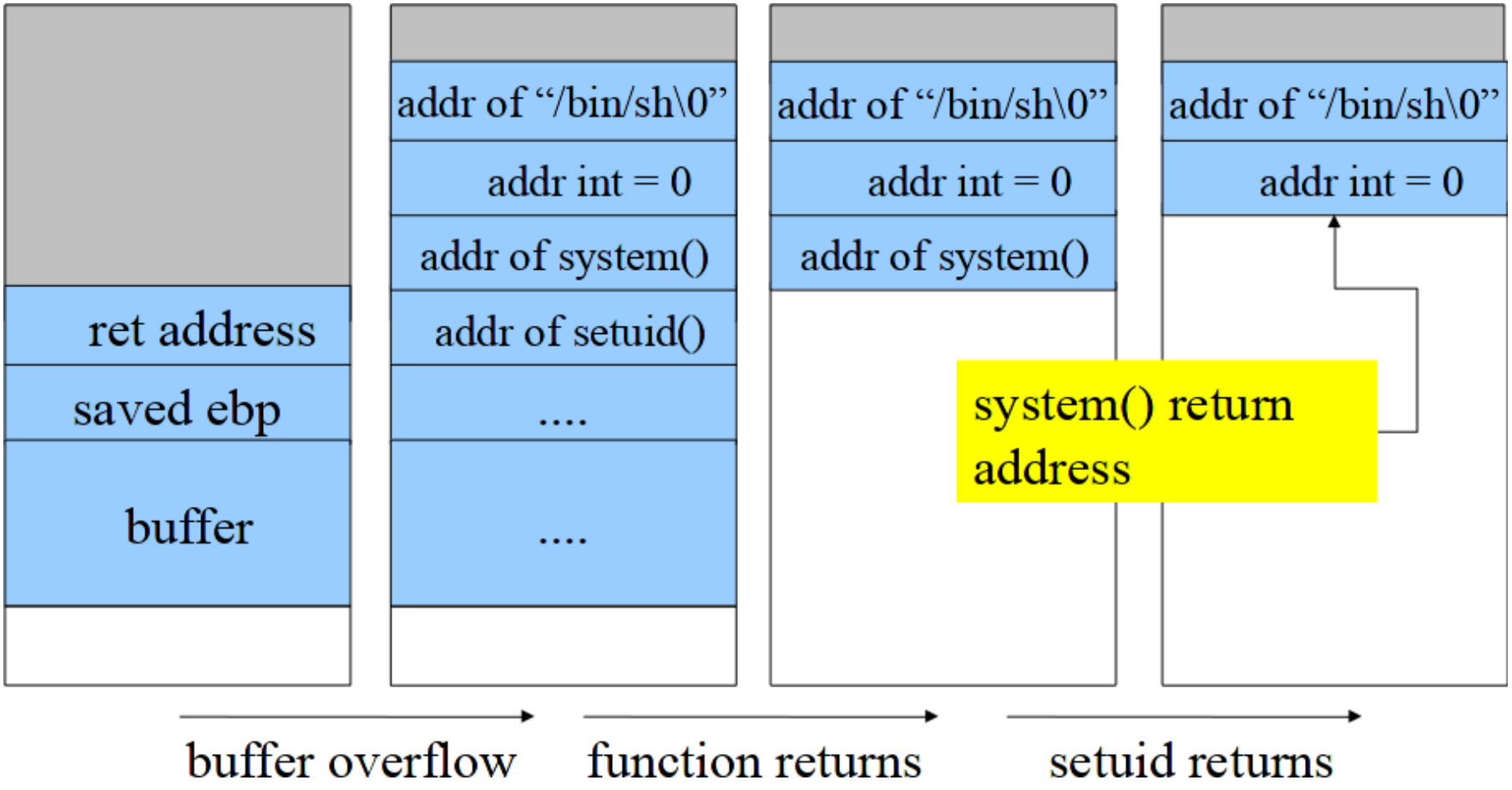
# Use libC to Move Shellcode (strcpy)





# Chain Function Calls





# Disadvantages of return-into-libc



- Functionality limited to loaded libraries and their functions
- No loops or conditional branches
- Solution: reuse code snippets instead of whole functions

# Return Oriented Programming (ROP)

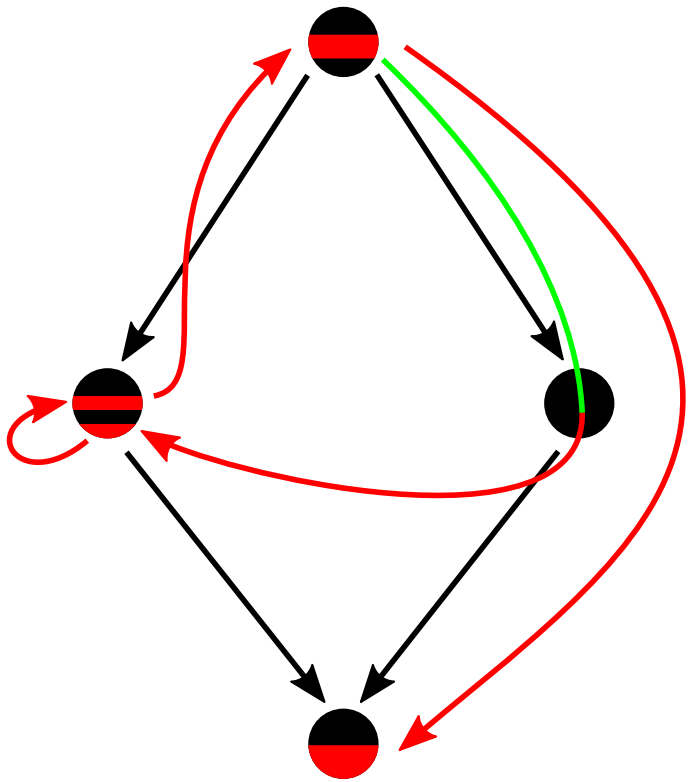
# What is ROP? - A Terminology

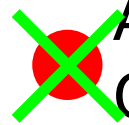


- Return Oriented Programming
  - A technique in exploitation to reuse existing code **gadgets** in a target binary as a method to bypass DEP
  - Also known as ROP
- Gadget
  - A sequence of meaningful instructions typically followed by a return instruction
  - Usually multiple gadgets are chained together to achieve desired functionality (like shellcode before)
  - These chains are called ROP Chains

# Return Oriented Programming - Concept







Attacker's Code

# Gadgets



- To build a ROP Chain you need gadgets

```
xor eax, eax
ret

pop ebx
pop eax
ret

add eax, ebx
ret
```

# ROPGadget –binary /bin/bash



```
0x0000000000457abc : xor edi, edi ; jmp 0x457aad
0x00000000004586f1 : xor edi, edi ; mov rax, rdi ; pop rbx ; ret
0x000000000045871b : xor edi, edi ; mov rax, rdi ; ret
0x00000000004b2e89 : xor edi, edx ; call qword ptr [rcx]
0x00000000004a9ba5 : xor edi, esp ; call qword ptr [rax]
0x00000000004023eb : xor edi, esp ; mov dh, 0xeb ; ret
0x0000000000427688 : xor edx, edx ; mov dword ptr [rax + 8], edx ; pop rbx ; ret
0x0000000000464798 : xor edx, edx ; mov rdi, rbp ; call 0x460a57
0x0000000000468290 : xor edx, edx ; test esi, esi ; sete dl ; jmp 0x46827e
0x000000000043f719 : xor edx, edx ; xor esi, esi ; mov edi, ebx ; call 0x43b9f8
0x00000000004a7a9d : xor esi, edi ; jmp qword ptr [rax]
0x000000000042acd1 : xor esi, esi ; mov ebx, edi ; call 0x416f66
0x000000000043f71b : xor esi, esi ; mov edi, ebx ; call 0x43b9f6
0x000000000046b889 : xor esp, esp ; call 0x4175b8
0x000000000048a721 : xor esp, esp ; mov eax, r12d ; pop rbx ; pop rbp ; pop r12 ; ret
0x0000000000438703 : xor esp, esp ; mov rax, r12 ; pop rbx ; pop rbp ; pop r12 ; ret
0x000000000046b888 : xor r12d, r12d ; call 0x4175b9
0x000000000045d0c6 : xor r13d, r13d ; call 0x489345

Unique gadgets found: 8653
┌─[atlas][/tmp]
└──╼ 
```

# How ROP Works



- Use gadgets to mimic functionality of shellcode (almost always possible)

**exit(0) - shellcode**

```
xor    eax, eax
xor    ebx,ebx
inc    eax
int    0x80
```

**exit(0) - ROP chain**

```
xor    eax, eax
ret
-----------------
xor    ebx,ebx
ret
-----------------
inc    eax
ret
-----------------
int    0x80
```

# How ROP Works

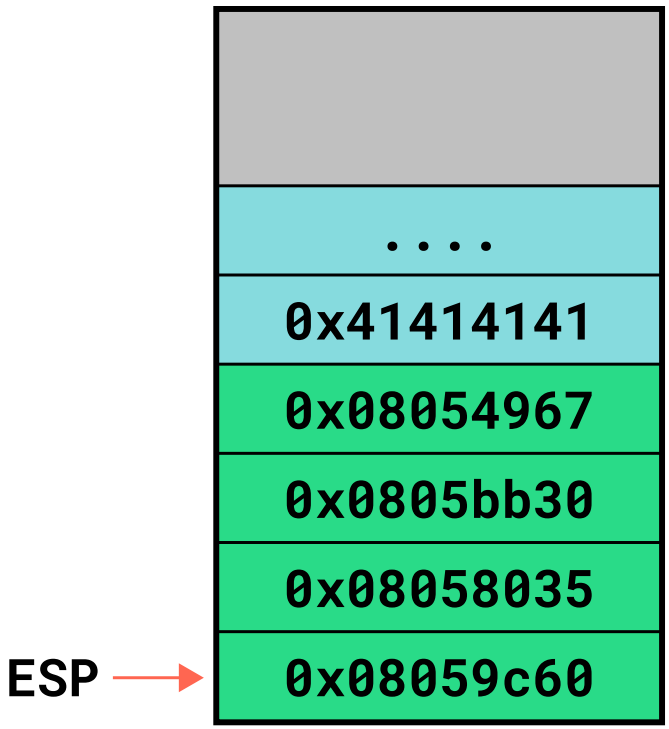


```
xor    eax, eax
ret
```

```
xor    ebx,ebx
ret
```

```
inc    eax
ret
```

```
int    0x80
```



# How ROP Works



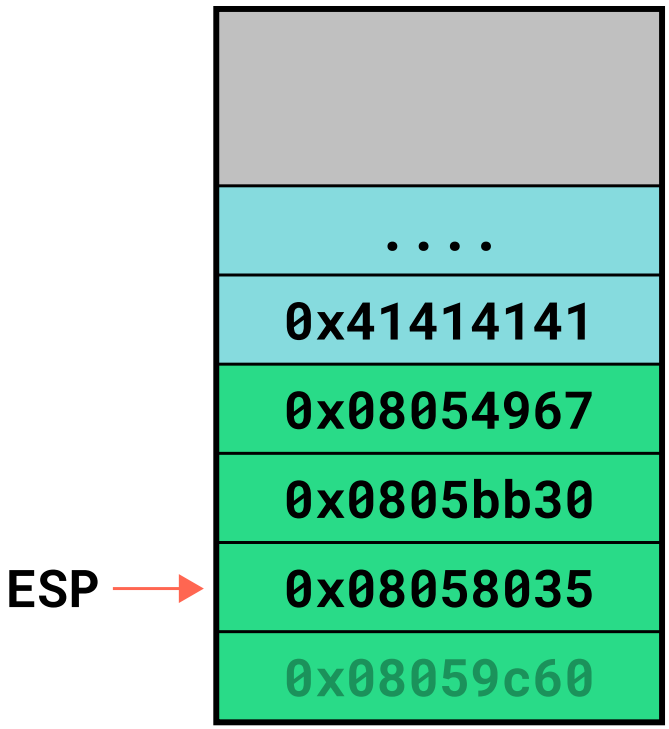
```
xor    eax, eax
ret
```

EIP

```
xor    ebx,ebx
ret
```

```
inc    eax
ret
```

```
int    0x80
```



# How ROP Works



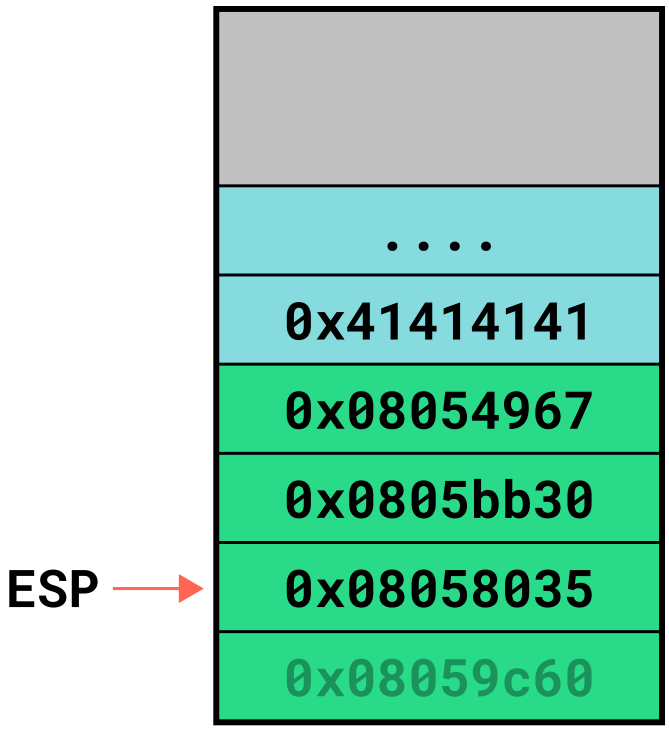
```
xor    eax, eax
ret
```

EIP

```
xor    ebx,ebx
ret
```

```
inc    eax
ret
```

```
int    0x80
```



# How ROP Works

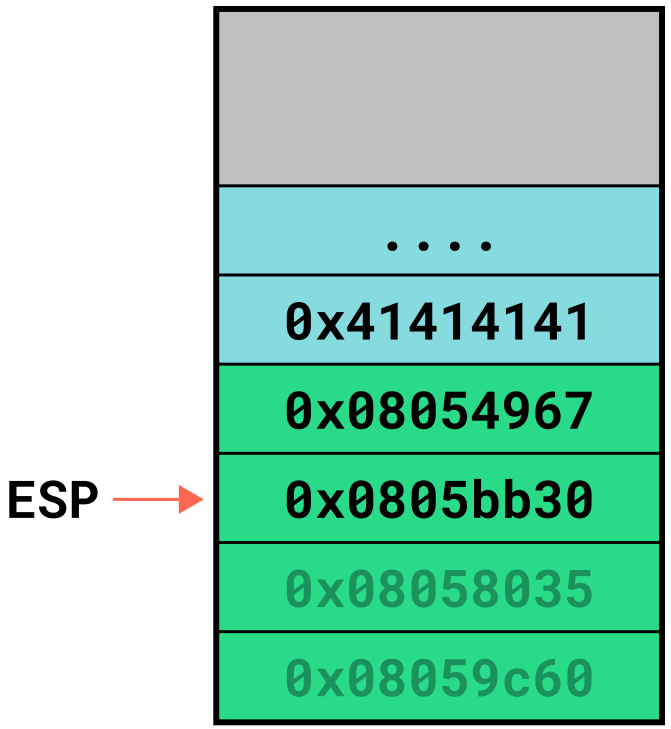


```
xor    eax, eax
ret
```

```
xor    ebx,ebx      ← EIP
ret
```

```
inc    eax
ret
```

```
int    0x80
```



# How ROP Works



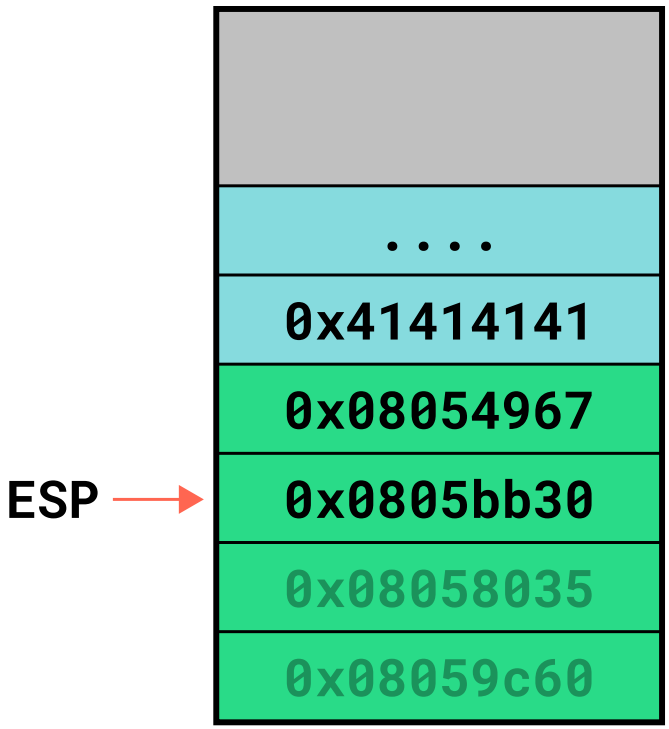
```
xor    eax, eax
ret
```

```
xor    ebx,ebx
ret
```

← EIP

```
inc    eax
ret
```

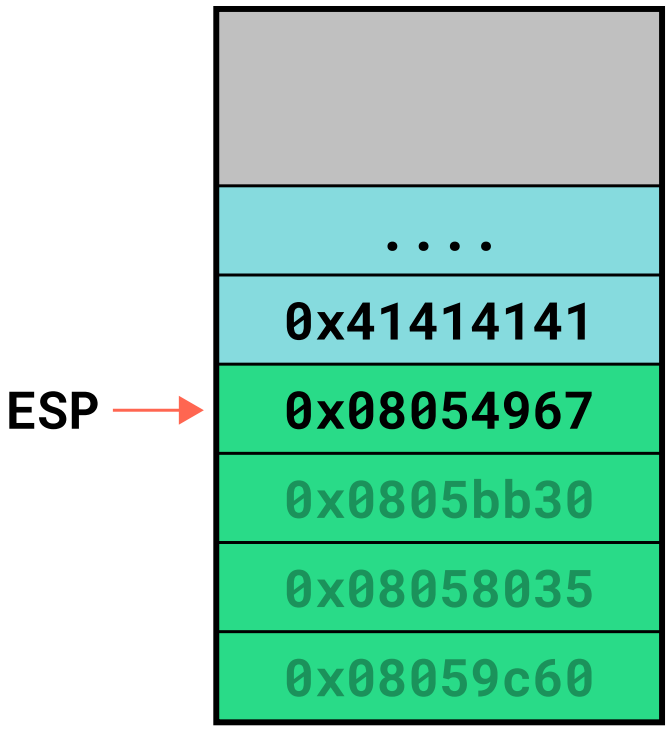
```
int    0x80
```



# How ROP Works



```
xor    eax, eax
ret
.....
xor    ebx,ebx
ret
.....
inc    eax          <-- EIP
ret
.....
int    0x80
```



# How ROP Works



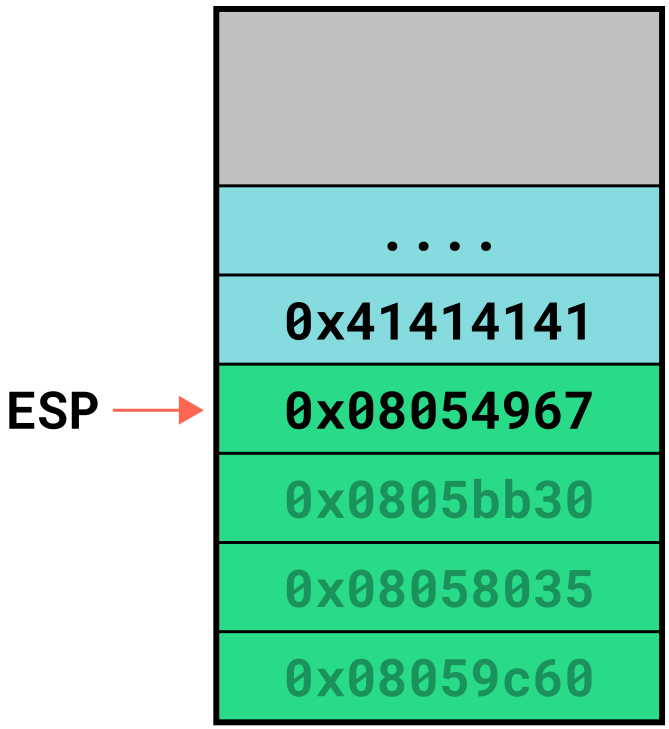
```
xor    eax, eax
ret
```

```
xor    ebx,ebx
ret
```

```
inc    eax
ret
```

← EIP

```
int    0x80
```



# How ROP Works



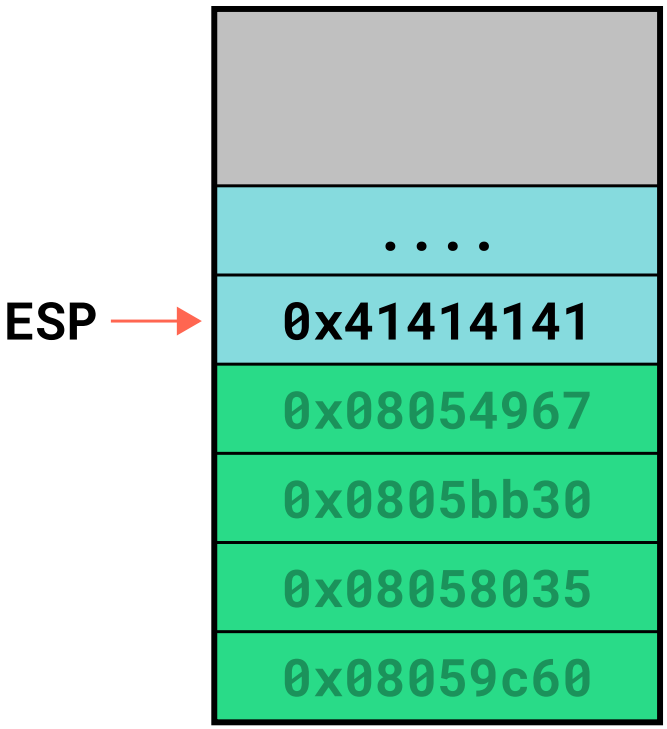
```
xor    eax, eax
ret
```

```
xor    ebx,ebx
ret
```

```
inc    eax
ret
```

```
int    0x80
```

EIP



# ROP Wrapup



- Called a function without the need to execute our own shellcode
- This was a very simple example for a ROP chain
  - Usually you will not find every gadget directly
  - Sometimes Gadget have side effects (e.g. pop a register)
  - In that case you will have to adjust the stack accordingly
    - e.g. put junk data on the stack, so that your chain will work as expected

# ASLR

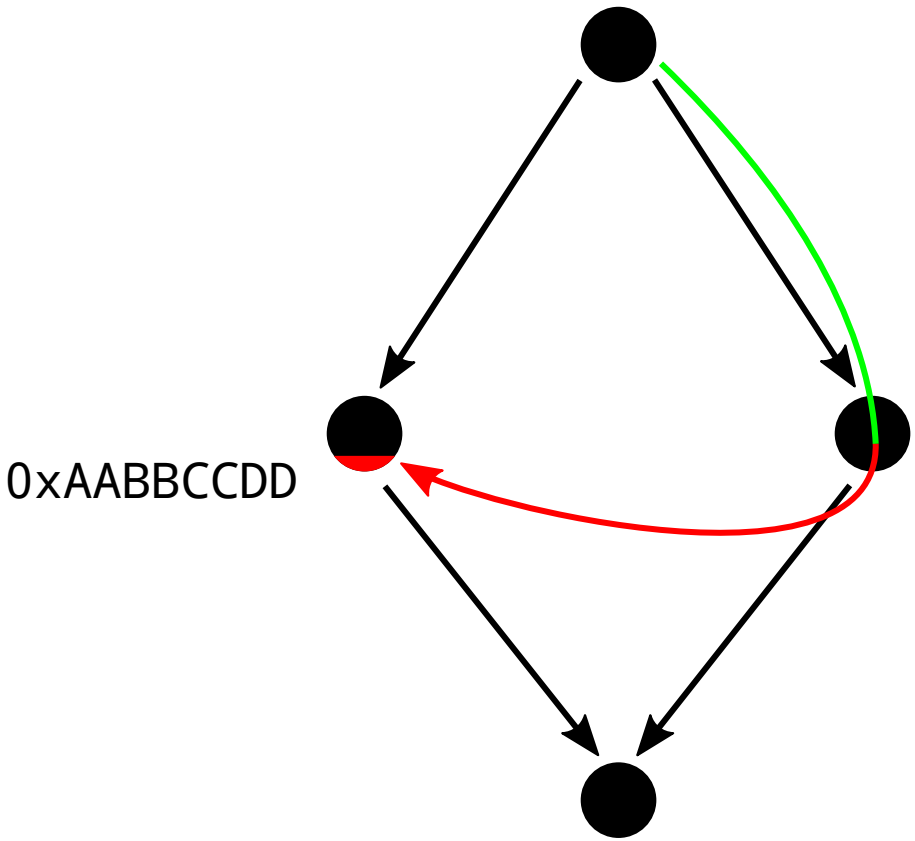


- Address Space Layout Randomisation
- Randomize Memory Layout
  - Make memory location of code and data unpredictable
- Binaries and libraries have to be compiled as PIE
  - Otherwise they will be at fixed locations again

# ASLR



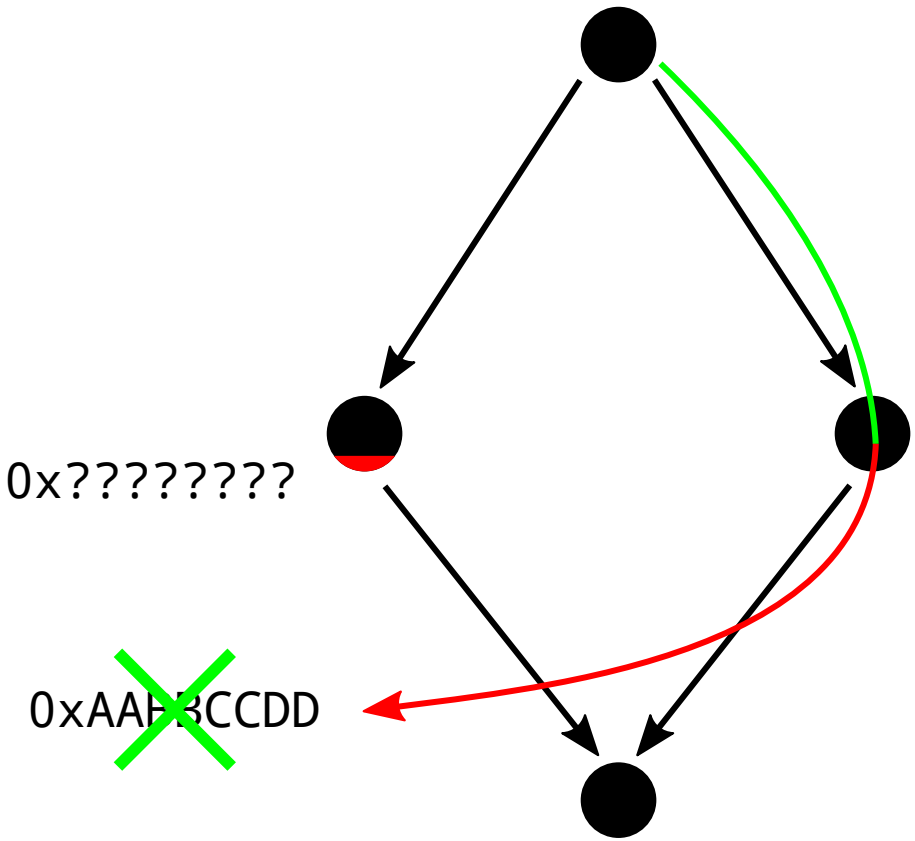
- ROP Attack without ASLR



# ASLR



- ROP Attack with ASLR



# Position Independent Executable (PIE)



- Programs assume to be always mapped to a certain base address
  - Important when e.g. absolute addressing modes are used
  - e.g. `jmp 0x400abcd`
- This is a big Problem for random remapping
  - Create code, which does not rely on absolute addresses
  - But uses only relative offsets

```
# gcc -pie -fPIE rip.c -o rip
```

# ASLR

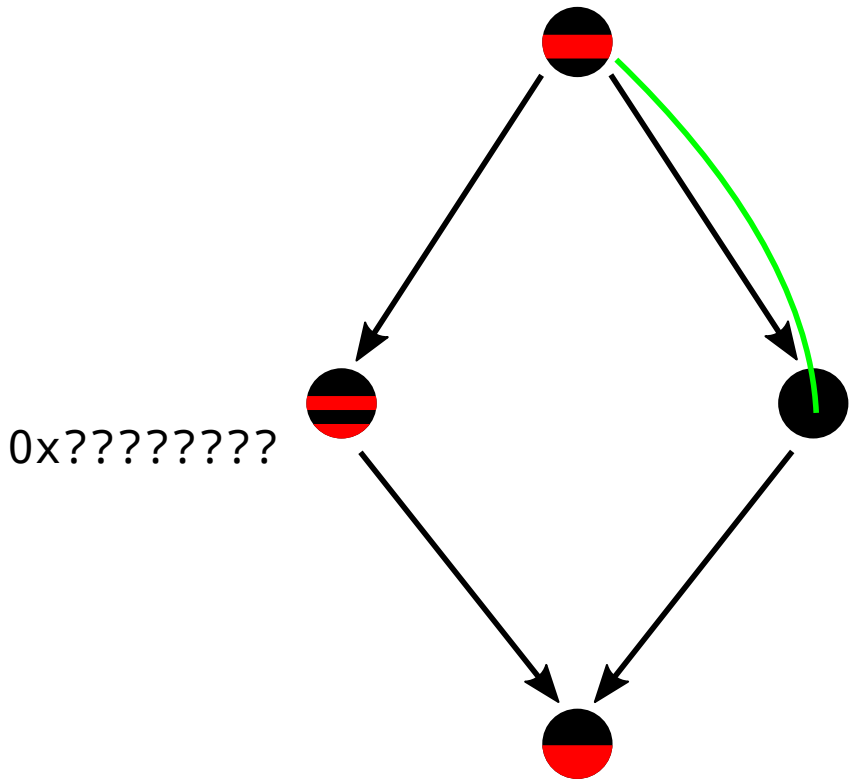


- What if everything can be/is randomized?
  - Still vulnerable to information leaks!
  - Code layout itself stays the same (even if not at the same base)
  - Therefore a single leak is enough to calculate offsets and start attack

# ASLR with information leak



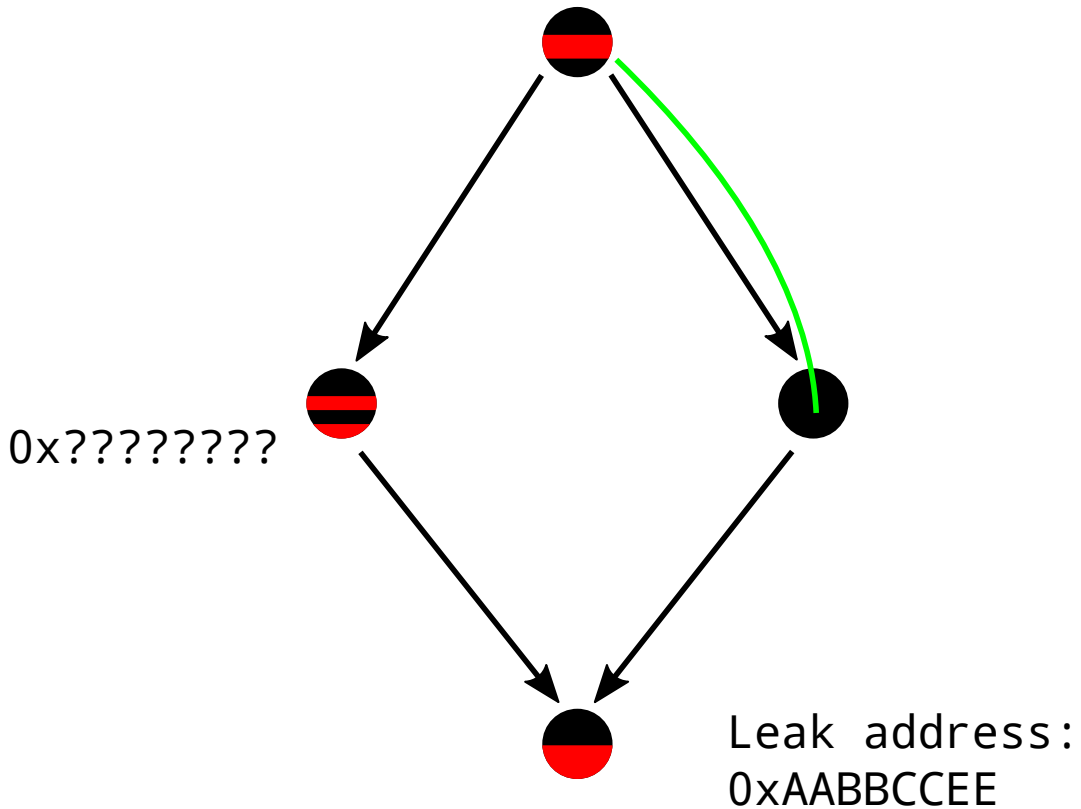
- Target address unknown



# ASLR with information leak



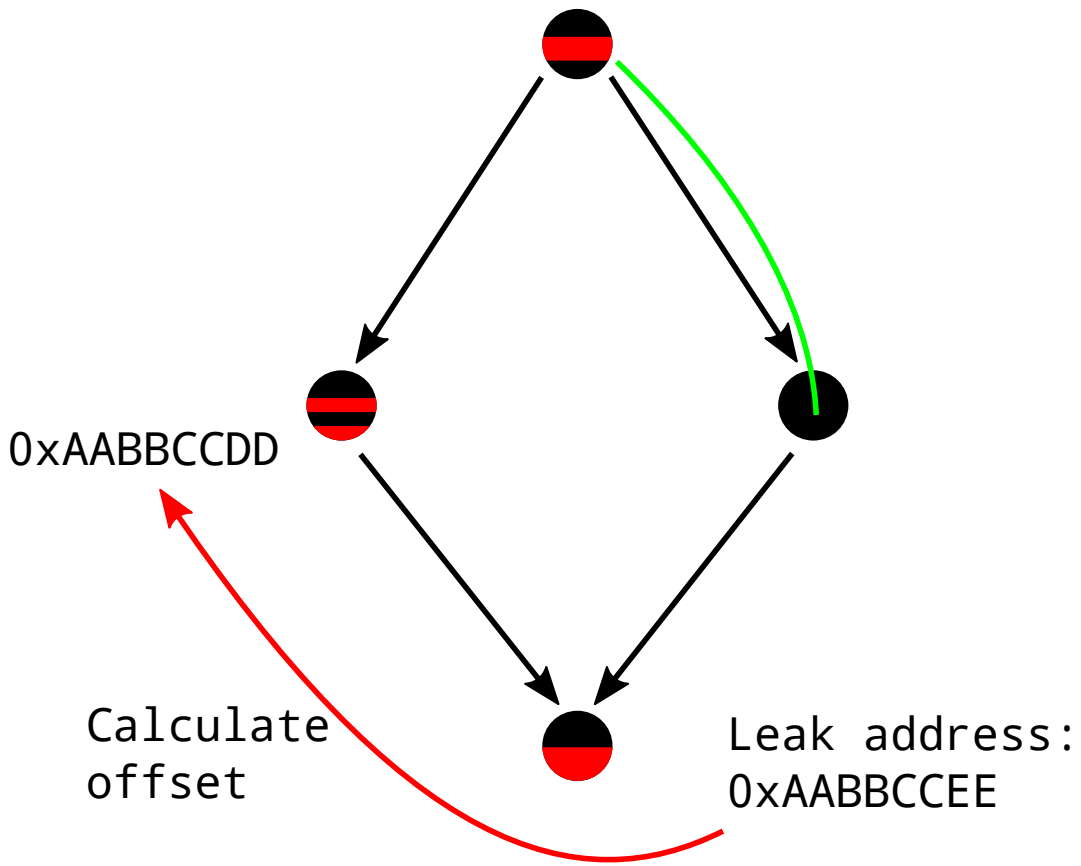
- Leak another address



# ASLR with information leak



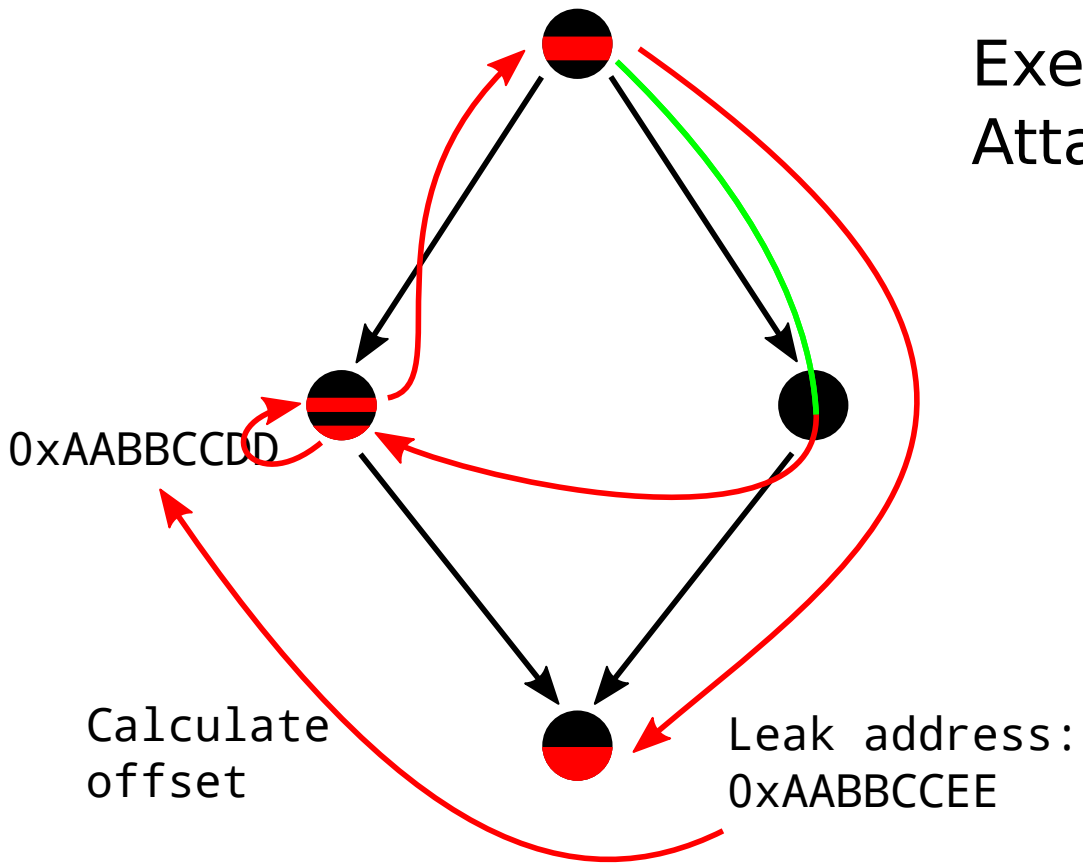
- Calculate offset



# ASLR with information leak



- Win!



# Summary



- Buffer Overflow Wrapup
- Mitigations
- Return Oriented Programming